CHAPTER

Fault simulation and test generation

14

James C.-M. Li
National Taiwan University, Taipei, Taiwan

Michael S. Hsiao
Virginia Tech, Blacksburg, Virginia

ABOUT THIS CHAPTER

Very large-scale integration (VLSI) circuits can be defective because of the imperfect manufacturing process. One of the most important tasks in VLSI testing is to minimize the number of defective chips shipped to customers. The quality of test patterns is critical in determining the thoroughness of testing. This requires the assessment of the quality of test patterns either developed manually or generated automatically so that a desired product quality can be achieved.

This chapter consists of two major VLSI testing topics: fault simulation and test generation. In fault simulation, we start with a discussion on fault collapsing. After an introduction of equivalent faults and dominant faults, the serial, parallel, concurrent, and differential fault simulation techniques are described, followed by qualitative comparisons between their advantages and drawbacks. These techniques trade accuracy for reduced execution time, which is crucial for managing the complexity of large designs. After fault simulation, basic ***automatic test pattern generation*** (ATPG) techniques, including Boolean difference, PODEM, and FAN, are described in detail. Advanced test generation techniques are also introduced to meet the demand for quality testing, including sequential ATPG, delay fault ATPG, and bridging fault ATPG. Throughout this chapter, the reader will learn about the major fault simulation and test generation techniques. This background will be valuable in selecting the test method that best meets the design needs and understands the relationship between test patterns and product quality.

14.1 INTRODUCTION

Simulation techniques have been widely used in VLSI designs for digital circuit verification, test development, design debug, and fault diagnosis. During the design stage, **logic simulation**, which has been extensively discussed in

Chapter 8, is performed to help verify whether the design meets its specifications and contains any design errors. It also helps locate design errors that may have escaped from detection during design debug.

Once the design meets its specifications and is ready for physical implementation, one must ensure that the manufactured devices will function as intended and no defective parts are shipped to customers. To achieve high product quality, typically with a defect level less than 500 ***defective parts per million*** (DPM), quality test patterns must be developed. At present, as we move to the nanometer design era, this has required applying **fault simulation** and ***automatic test pattern generation*** (ATPG) to the design that has been embedded with ***design for testability*** (DFT) features during test development.

In contrast to logic simulation, **fault simulation** is used to measure the effectiveness of test patterns in detecting defects that might have been introduced during the manufacturing process. This requires simulating the faulty behavior of the circuit in detecting the modeled faults of interest. (For this reason, logic simulation is generally referred to as **fault-free simulation**.) Furthermore, fault simulation is an integral component of any ATPG program.

The major difference between logic simulation and fault simulation lies in the nature of the non-idealities they deal with. Logic simulation is intended for checking whether the circuit's responses to a given set of input vectors conform to the given specifications or a known good design as the reference. Design errors may be introduced by human designers or ***electronic design automation*** (EDA) tools, and they should be caught before physical implementation. **Fault simulation**, on the other hand, is concerned with checking the behavior of fabricated circuits as a consequence of inevitable fabrication process imperfections. The manufacturing defects (*e.g.*, wire shorts and opens), if present, may cause the circuits to behave differently from the expected behavior. Fault simulation generally assumes that the design is functionally correct (*i.e.*, free of design errors), and it is targeted at capturing manufacturing defects. However, we note that fault simulation methods may be applied during the design verification stage as well.

The capability of fault simulation to predict the faulty circuit behavior is of great importance for test and diagnosis. First, fault simulation evaluates the effectiveness of a set of test patterns in detecting manufacturing defects. The quality of a test set is expressed in terms of **fault coverage**, which is the ratio of detected faults to the total number of faults in the circuit. In practice, the designer uses a **fault simulator** to evaluate the fault coverage of a set of input stimuli (test vectors or test patterns) with respect to the modeled faults of interest. Because fault simulation concerns the fault coverage of a test set rather than the detection of design bugs, it is also termed **fault grading**. Low fault coverage test patterns will jeopardize the manufacturing test quality and eventually lead to unacceptable field returns from customers. Second, fault simulation helps to identify undetected faults, which is especially important when the achieved fault coverage is below an acceptable level. In this case, either the designer or

the ATPG has to generate additional test vectors to improve the fault coverage (*i.e.*, to detect those remaining undetected faults). Third, as part of the **test compaction** process, fault simulation identifies redundant test patterns, which may be discarded with no negative impact on the fault coverage. With the preceding capabilities and applications, fault simulation is one of the crucial components of ATPG. In fact, the implementation of an ATPG program usually starts with the fault simulator. Finally, fault simulation assists **fault diagnosis**, which determines the type and location of faults that best explain the faulty circuit behavior of the device under diagnosis. The fault simulation results are compared with the observed circuit responses to identify the most likely faults. The fault type and location information can then be used as a starting point for locating the defects that cause the circuit malfunction.

Although logic and fault simulators can provide important information about the behavior of the circuit, they require a set of test vectors with which the circuit is simulated. The objective of test generation, then, is the task of producing a set of test vectors that will uncover any defect in a chip. Figure 14.1 illustrates a high-level concept of test generation. In this figure, the circuit at the top is defect free, and for any defective chip that is functionally different from the defect-free one there must exist some input that can differentiate the two. Generating effective test patterns efficiently for a digital circuit is thus the goal of an ATPG system.

Because this problem is extremely difficult, DFT methods have been frequently used to relieve the burden on the ATPG. In this sense, a powerful ATPG can be regarded as the "Holy Grail" in testing, with which all DFT methods could potentially be eliminated. In other words, if the ATPG engine is capable of efficiently delivering high-quality test patterns that achieve high fault coverages and small test sets on large, complex chips, DFT would no longer be necessary.

Because fault simulation can help to determine those faults that could be detected by the same generated test, it becomes an essential component of ATPG. By removing those incidentally detected faults, ATPG is able to significantly

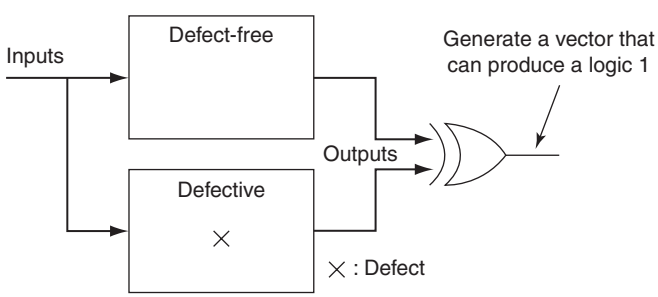


FIGURE 14.1

Conceptual view of test generation.

reduce the number of faults that it needs to consider after the generation of each new test vector, thereby improving the efficiency of the ATPG process.

For some fault models, the circuit layout information is needed. For example, wire delay values are needed to compute the longest paths, and the actual positions of gates and wires are needed to identify those likely bridges. However, because ATPG is a time-consuming process, we would like to start the ATPG process before the layout is available. In this regard, an ATPG may be performed to obtain an initial test set without the layout information. Then, after **place and route**, any faults that require circuit layout information that are undetected by the test set would be identified, and the ATPG can be invoked again to target these specific undetected faults to ensure test quality.

14.2 FAULT COLLAPSING

Fault collapsing reduces the number of faults to be considered in fault simulation and ATPG so the overall run time can be reduced. Two requirements must be met for fault collapsing to become effective. First, fault collapsing must run much faster than fault simulation or ATPG; otherwise, fault collapsing may not be worth doing. Second, the collapsed faults must be representative of all original faults modeled in the circuit. In this section, we introduce two fault-collapsing techniques: **equivalence fault collapsing** and **dominance fault collapsing**. Linear time algorithms are given to meet the first requirement. We illustrate that dominance fault collapsing produces a fewer number of faults than equivalence fault collapsing. However, from a fault coverage accuracy viewpoint, equivalence fault collapsing is more often quoted than dominance fault collapsing, because the former results in a better indication of the test quality.

14.2.1 Equivalence fault collapsing

Let two faults f and g be said to be **functionally equivalent** (or simply **equivalent**) if the faulty outputs of these two faults are identical for any input [McCluskey 1971; Abramovici 1994; Bushnell 2000]. Equivalent faults are **indistinguishable**, because there is no test pattern that can tell them apart. Consider the example of a two-input AND gate shown in Figure 14.2. The good outputs and faulty outputs of the AND gate for all four possible input combinations are listed in Table 14.1. From this table, we can see that A stuck-at zero fault (denoted as $A/0$) and C stuck-at zero fault (denoted as $C/0$) are equivalent.

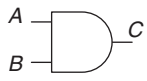


FIGURE 14.2

An example two-input AND gate.

Table 14.1 Good and faulty outputs for Figure 14.2

Input		Output						
A	*B*	*C*	*A*/0	*C*/0	*B*/0	*A*/1	*C*/1	*B*/1
0	0	0	0	0	0	0	**1**	0
0	1	0	0	0	0	**1**	**1**	0
1	0	0	0	0	0	0	**1**	**1**
1	1	1	**0**	**0**	**0**	1	1	1

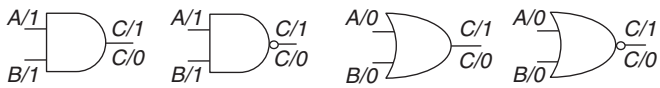


FIGURE 14.3

Equivalence collapsed fault list for four elementary gates.

This is because the faulty outputs of these two faults are always the same for all the four input combinations. On the other hand, the A stuck-at one fault ($A/1$) and the C stuck-at one fault ($C/1$) are not equivalent, because the input pattern $A = B = 0$ can distinguish these two faults. Another input pattern is $A = 1$ and $B = 0$. For clear illustration, the faulty outputs that are different from good outputs are underlined and highlighted in bold.

The equivalence relationship is **symmetric**. This means, if fault f is equivalent to fault g, then fault g is equivalent to fault f. The equivalence relationship is also **transitive**. That is, if fault f is equivalent to fault g and fault g is equivalent to fault h, then fault f is equivalent to fault h. For the example given in Figure 14.2, $A/0$ fault is equivalent to $B/0$ fault, and $B/0$ fault is equivalent to $C/0$ fault. These three faults $\{A/0, B/0, C/0\}$ belong to the same **equivalence class**.

Equivalence fault collapsing reduces the set of faults that needs to be considered with the fault equivalence relation. Only one representative fault is selected from every equivalent class. Figure 14.3 shows the **equivalence collapsed fault list** for four types of elementary gates. Originally, there are six faults associated with a two-input AND gate: $A/1$, $A/0$, $B/0$, $B/1$, $C/0$, and $C/1$. After equivalence fault collapsing, the number of faults is reduced to four: $A/1$, $B/1$, $C/1$, and $C/0$. The other types of gates can be examined in the same way. Generally speaking, an n-input elementary gate has $2n$ and $n + 2$ stuck-at faults before and after equivalence fault collapsing, respectively. Note that the equivalence fault collapsed fault list is not unique, and there are other ways to collapse the faults than are shown in Figure 14.3. For example, $\{A/0, A/1, B/1, C/1\}$ is another possible way to perform equivalence fault collapsing.

Equivalence fault collapsing can be performed by either functional analysis or structural analysis. Exhaustive functional analysis is time-consuming, because enumeration of 2^n patterns may be needed for an n-input circuit (like Table 14.1 for the AND gate shown in Figure 14.2). Therefore, in the following text, we only demonstrate a linear-time structural analysis to perform equivalence fault collapsing. The resulting equivalence collapsed fault list may not be minimal, but structural analysis is good enough for most applications.

For a fanout-free circuit consisting of elementary gates (such as buffers, inverters AND, OR, NAND, and NOR gates), equivalence fault collapsing can be performed by keeping two kinds of faults: (1) both stuck-at one and stuck-at zero faults on every primary output, and (2) one collapsed fault on each gate input whose stuck value is shown in Figure 14.3. Inverters and buffers should be treated as wires. For the example in Figure 14.4, we keep both $H/0$ and $H/1$ faults on primary output H. We also keep one fault on each gate input, such as $A/0$ and $B/0$ for OR gate G_1, etc. Note that faults on the gate outputs are removed, because they are equivalent to some other faults in the figure. For example, gate G_1 output stuck-at zero fault is equivalent to $C/0$ fault, which is again equivalent to $E/1$ fault, which is in turn equivalent to $H/0$ fault.

For circuits with fanouts, fault collapsing becomes complicated, because faults on the fanout stem are now always equivalent to the faults on the fanout branches. Figure 14.5 shows a circuit with a fanout stem E and two fanout branches L and F. According to Table 14.2, $E/0$ fault is equivalent to $F/0$ fault but not equivalent to $L/0$ fault. Also, none of the stuck-at one faults are equivalent. **Stem analysis** is required to determine equivalent faults on a fanout stem and its branches. However, stem analysis is generally not cost-effective in terms of CPU time, so the details are skipped in this chapter.

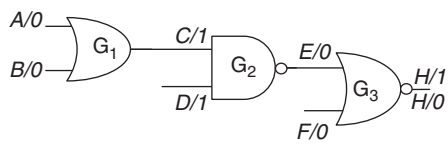


FIGURE 14.4

Equivalence fault collapsing on a fanout-free circuit.

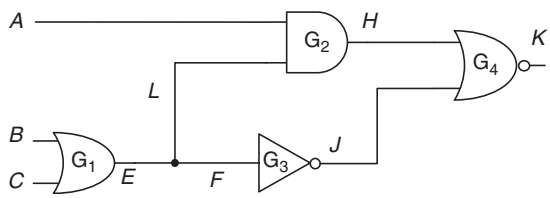


FIGURE 14.5

Equivalence fault collapsing for faults on fanouts.

Table 14.2 Good and faulty outputs for Figure 14.5

Input			Output						
A	*B*	*C*	*E*	*E*/0	*F*/0	*L*/0	*E*/1	*F*/1	*L*/1
0	0	0	0	0	0	0	1	1	0
0	0	1	1	0	0	1	1	1	1
0	1	0	1	0	0	1	1	1	1
0	1	1	1	0	0	1	1	1	1
1	0	0	0	0	0	0	0	1	0
1	0	1	0	0	0	1	0	0	0
1	1	0	0	0	0	1	0	0	0
1	1	1	0	0	0	1	0	0	0

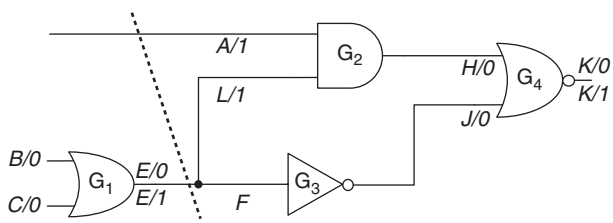


FIGURE 14.6

Equivalence collapsed fault list for Figure 14.5.

To avoid stem analysis, an approximation solution is used to partition the circuit into independent ***fanout-free regions*** (FFRs). Every fanout stem is treated as a primary output, so both stuck-at one and stuck-at zero faults are included in the collapsed fault list. Algorithm 14.1 introduces a simple equivalence fault-collapsing (simple_EFC) algorithm without stem analysis. Figure 14.6 shows the resulting equivalence collapsed fault list with the simple_EFC algorithm. The circuit is partitioned into two independent fanout-free regions: four faults in one region and six faults in the other. The simple_EFC algorithm reduces the number of faults from 18 to 10. Please note that inverter G_3 is ignored in this algorithm, because its input stuck-at one fault is always equivalent to its output stuck-at zero fault and *vice versa*. Also note that simple_EFC is not the only way to perform fault collapsing; other implementations of fault collapsing are possible.

Algorithm 14.1 A simple equivalence fault-collapsing algorithm

```
simple_EFC (N) /*N is a netlist*/
 1. fault_list = {};
 2. foreach gate or PO or PI g in N
 3.   if ((g is PO) || (g is PI and fanout stem)) then
 4.     fault_list = fault_list ∪ g stuck-at 0 and 1;
 5.   else if (output of gate g is fanout stem) then
 6.     fault_list = fault_list ∪ g output stuck-at 0 and 1;
 7.   end if
 8.   if (gate g is AND) || (gate g is NAND) then
 9.     fault_list = fault_list ∪ g input stuck-at 1;
10.   else if (gate g is OR) || (gate g is NOR) then
11.     fault_list = fault_list ∪ g input stuck-at 0;
12.   end if
13. end foreach
14. return (fault_list);
```

The simple_EFC algorithm can complete in linear time because it checks every gate exactly once. However, this algorithm has two drawbacks. First, the result is not optimal, because it lacks stem analysis. For example, Table 14.2 shows that *E*/0 fault is actually equivalent to *K*/0 fault, but they both appear in Figure 14.6. This small error, however, is often acceptable in most cases. Second, the relationship between the original (uncollapsed) faults and the corresponding collapsed faults is lost. For example, the link is lost between the four faults {*F*/0, *J*/1, *H*/1, *K*/0} in the same **equivalence class** and their collapsed fault *K*/0. The relation between **uncollapsed faults** and **collapsed faults** is needed when calculating the fault coverage of the circuit. Fault coverage can be calculated on the basis of either the uncollapsed faults or the collapsed faults. The **uncollapsed fault coverage** is the number of detected uncollapsed faults over the total number of uncollapsed faults, whereas the **collapsed fault coverage** is the number of detected collapsed faults over the total number of collapsed faults. Oftentimes, these two numbers are not identical but close to each other. Missing the link between the collapsed faults and the uncollapsed fault makes it difficult to convert the collapsed fault coverage to the uncollapsed fault coverage. However, modern fault simulators and ATPG programs have found an easy way to rebuild the link by performing another pass of linear-time analysis on equivalent faults.

14.2.2 Dominance fault collapsing

The equivalence collapsed fault list can be further compressed with the **fault dominance** relationship. Let the **detecting set** of fault f (denoted as T_f) be the set of all test patterns that detect fault f. Fault f dominates fault g if the

detecting set of fault f contains that of fault g. That means, $T_f \supseteq T_g$. For the example in Figure 14.2, fault $C/1$ dominates fault $A/1$ because the detecting set of $C/1$ {00, 01, 10} contains the detecting set of $A/1$ {01}. The dominance relation is not symmetric but is transitive.

If a test pattern detects the **dominated fault**, then it must detect the corresponding **dominating fault**. To reduce the run time, the dominating faults can be removed from the fault list. The reduction of fault list with the fault dominance relation is called **dominance fault collapsing**. If two faults are equivalent, then they dominate each other. Therefore, the number of dominance-collapsed faults must be smaller or equal to that of equivalence-collapsed faults.

Figure 14.7 shows the **dominance collapsed fault list** of four elementary gates. Originally, there are four equivalence-collapsed faults for a two-input AND gate: $A/1$, $B/1$, $C/0$, and $C/1$. After dominance fault collapsing, the number of faults is reduced to three: $A/1$, $B/1$, and $C/0$. The other types of gates can be examined in the same way. Generally speaking, for an n-input elementary gate, there are $n + 1$ stuck-at faults after dominance fault collapsing.

For a fanout-free circuit consisting of elementary gates (such as buffers, inverters AND, OR, NAND, and NOR gates), dominance fault collapsing can be performed according to the following two rules: (1) one collapsed fault on every primary input whose value is shown in Figure 14.7, and (2) one collapsed fault on each gate output whose gate inputs are all primary inputs. Those gates whose inputs are all primary inputs are called **input gates**. Inverters and buffers should be treated as wires. Figure 14.8 shows the dominance collapsed fault list of the example fanout-free circuit. Note that no fault is needed on G_2 gate output, because G_2 is not an input gate. $E/0$ fault dominates $C/1$ fault, so the former can be removed. *E/1* fault is equivalent to *C/0* fault, which dominates *A/0* fault, so both *C/0* and *E/1* faults are removed. The explanation of the other faults is similar so it is left as an exercise for the readers. This circuit has 14 uncollapsed faults, which are reduced to 8 equivalent faults and then to 5 dominant faults after equivalence and dominance fault collapsing, respectively.

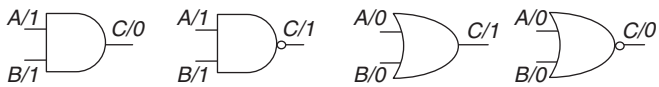


FIGURE 14.7

Dominance collapsed fault list for four elementary gates.

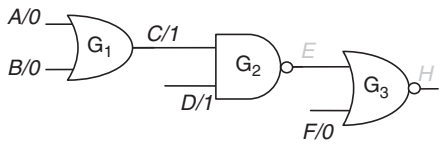


FIGURE 14.8

Dominance fault collapsing on a fanout-free circuit.

Faults on the fanout branches do not always dominate faults on the fanout stem. Consider again the example in Figure 14.5. According to Table 14.2, *F*/1 fault dominates *E*/1 fault. However, *L*/1 fault does not dominate *E*/1 fault. (Actually, *L*/1 fault has an empty detecting set so *L*/1 is a **redundant fault**. More details on redundant faults are given in the test generation section.) Again, stem analysis is needed to determine whether fanout branch faults dominate fanout stem faults.

An approximation method to avoid stem analysis is to partition the circuit into fanout-free regions and perform fault collapsing on each fanout-free region independently. A simple_DFC algorithm is shown in Algorithm 14.2. The dominance fault collapsed result is shown in Figure 14.9. The number of faults is reduced to seven. Without stem analysis, the result is not optimal because *J*/0 is equivalent to *F*/1, which dominates *E*/1.

Algorithm 14.2 A simple dominance fault-collapsing algorithm

```
simple_DFC (N) /*N is a netlist*/
 1. fault_list = {};
 2. foreach gate or PI or PO g in N
 3.   if ((g is PI and fanout stem) || (g is PO and fanout branch)) then
 4.     fault_list = fault_list ∪ g output stuck-at 0 and 1;
 5.   else if (g is gate) then
 6.     foreach gate input i of gate g
 7.       h = backtrace inverters starting from i;
 8.       if (h is PI or fanout branch) then /* rule #1 */
 9.         if (gate g is AND) || (gate g is NAND) then
10.           fault_list = fault_list ∪ i stuck-at 1;
11.         else if (gate g is OR) || (gate g is NOR)
12.           fault_list = fault_list ∪ i stuck-at 0;
13.         end if
14.       end if
15.     end foreach
16.     if (every input of g has a fault) then /* rule #2 */
17.       if (gate g is AND) || (gate g is NOR) then
18.         fault_list = fault_list ∪ g output stuck-at 0;
19.       else if (gate g is OR) || (gate g is NAND) then
20.         fault_list = fault_list ∪ g output stuck-at 1;
21.       end if
22.     end if
23.   end if
24. end foreach
25. return (fault_list);
```

Although the dominance collapsed fault list is smaller than the equivalence collapsed fault list, fault coverage of the former is not as representative as that of the latter. The reason is that a test pattern may detect a dominating fault without

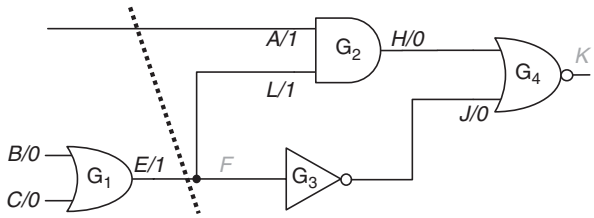


FIGURE 14.9

Dominance collapsed fault list for Figure 14.5.

detecting the dominated fault. For the example given in Figure 14.5, test pattern ABC = 100 does not detect the dominated fault *E*/1 but it detects the dominating fault *F*/1. If the dominance collapsed fault list is used during fault simulation, the dominance collapsed fault coverage may underestimate the test quality. As a result, modern fault simulators and ATPG programs favor the use of equivalence fault collapsing only.

14.3 FAULT SIMULATION

Fault simulation is a more challenging task than logic simulation because of the added dimension of complexity (*i.e.*, the behavior of the circuit containing all the modeled faults must be simulated). When simulating one fault at a time, the amount of computation is approximately proportional to the circuit size, the number of test patterns, and the number of modeled faults. Because the number of modeled faults are roughly proportional to the circuit size, the overall time complexity of fault simulation is $O(pn^2)$, for p test patterns and n logic gates, which becomes infeasible for large circuits. To improve fault simulation performance, various fault simulation techniques have been developed. In this section, we restrict our discussion to the single stuck-at fault model and illustrate the key fault simulation techniques along with qualitative comparisons between their advantages and drawbacks.

14.3.1 Serial fault simulation

Serial fault simulation is the simplest fault simulation technique. It consists of fault-free and faulty circuit simulations. Initially, fault-free logic simulation is performed on the original circuit to obtain the fault-free output responses. The fault-free responses are stored and later used to determine whether a test pattern can detect a fault or not. After fault-free simulation, a serial fault simulator simulates faults one at a time. For each fault, **fault injection** is first performed,

which modifies the original circuit to mimic the circuit behavior in the presence of the fault. Then, the faulty circuit is simulated to derive the *faulty* responses of the current fault with respect to the given test patterns. This process repeats until all faults in the fault list have been simulated.

The serial fault simulation process is demonstrated with the example circuit N. In this example, the fault list comprises two faults, A stuck-at one (denoted by f) and J stuck-at zero (denoted by g), which are depicted in Figure 14.10. Note that, although both faults are drawn in the figure, only one fault is present at a time under the single stuck-at fault model. The test set consists of three test patterns (denoted by P_1, P_2, P_3, respectively, and shown in the "Input" columns of Table 14.3).

The serial fault simulator starts from fault-free simulation. The fault-free responses are $K = \{1, 1, 0\}$ for input patterns P_1, P_2, and P_3, respectively. After the fault-free responses are available, fault f is processed—fault injection is achieved by forcing A to a constant one, and the obtained faulty circuit is simulated. The circuit responses for fault f are $K_f = \{0, 0, 0\}$ with respect to the three input patterns. Compared with the fault-free responses (the "Output" column in Table 14.3), it is observed that patterns P_1 and P_2 detect fault f, but pattern P_3 does not. After fault f has been simulated, circuit N is restored by removing fault f. The next fault g is then injected by forcing J to zero. Simulation of the resulting faulty circuit is then performed to obtain the faulty outputs $K_g = \{1, 1, 1\}$ (also listed in Table 14.3). Fault g is detected by pattern P_3 but not P_1 and P_2.

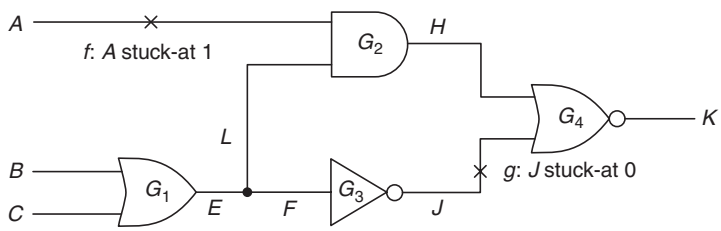


FIGURE 14.10

An example circuit with two faults.

Table 14.3 Serial Fault Simulation Results for Figure 14.10

Pat. #	Input			Internal					Output		
	A	B	C	E	F	L	J	H	K_{good}	K_f	K_g
P_1	0	1	0	1	1	1	0	0	1	0	1
P_2	0	0	1	1	1	1	0	0	1	0	1
P_3	1	0	0	0	0	0	1	0	0	0	1

In this example, nine simulation runs are performed: three fault-free and six faulty circuit simulations. These nine simulation runs can be divided into three **simulation passes**. In each simulation pass, either the fault-free or the faulty circuit is simulated for the whole test pattern set. Thus, the first simulation pass consists of fault-free simulations for P_1, P_2, and P_3, and the second and third passes correspond to the faulty circuit simulations of faults f and g, respectively, for P_1, P_2, and P_3.

By careful inspection of the simulation results in Table 14.3, one can observe that if we are only concerned with the set of faults that are detected by the test set $\{P_1, P_2, P_3\}$, simulations of the faulty circuit with fault f for patterns P_2 and P_3 are redundant, because f is already detected by P_1. (It is assumed that the test patterns are simulated in the order P_1, P_2, and then P_3.) Halting simulation of detected faults is called **fault dropping**. For the purpose of fault grading, fault dropping dramatically improves fault simulation performance, because most faults are detected after relatively few test patterns have been applied. Fault dropping, however, should be avoided in fault diagnosis applications in which the entire fault simulation results are usually required to facilitate the identification of the fault type and location.

The simplified serial fault simulation flow is depicted in Figure 14.11. Before fault simulation, *fault collapsing* is executed to reduce the size of the fault list, denoted by F. Fault-free simulation is then performed for all test patterns to obtain the correct responses O_g. The algorithm then proceeds to fault simulation. For each fault f in F, if there exists a test pattern whose output response O_f differs from that of the corresponding good circuit O_g, f is removed from F, indicating that it is detected. When all patterns have been simulated, the remaining faults in F are the undetected faults.

The major advantage of serial fault simulation is its ease of implementation—a regular logic simulator plus fault injection and output comparison procedures will suffice. In addition, serial fault simulation can handle a wide range of fault models, as long as the fault effects can be properly injected into the circuit. The major disadvantage of serial fault simulation is its low performance. As will be discussed in the following subsections, practical fault simulation techniques exploit parallelism and/or similarities among the faulty circuits to speed up the fault simulation process.

14.3.2 Parallel fault simulation

Similar to parallel logic simulation, fault simulation can take advantage of the bitwise parallelism inherent in the host computer to reduce fault simulation time. For instance, in a 32-bit wide CPU, logic operations (AND, OR, or XOR) can be performed on all 32 bits at once. There are two ways to realize bitwise parallelism in fault simulation: parallelism in faults and parallelism in patterns. These two approaches are referred to as **parallel fault simulation** and **parallel pattern fault simulation**.

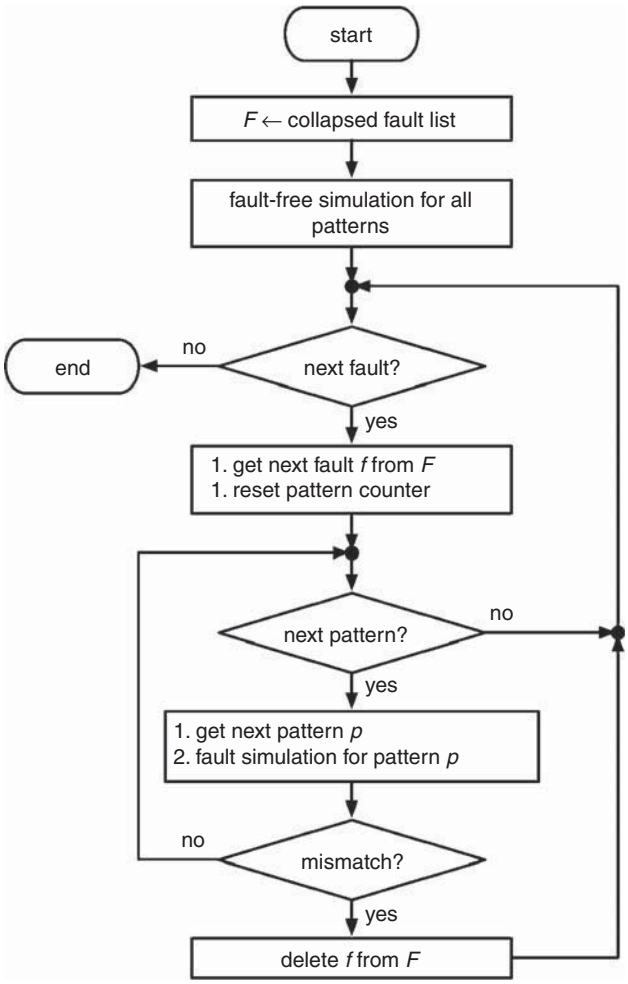


FIGURE 14.11

The serial fault simulation algorithm flow.

14.3.2.1 *Parallel fault simulation*

Parallel fault simulation was proposed in the early 1960s [Seshu 1965]. Assuming that binary logic is used, one bit is sufficient to store the logic value of a signal. Thus, in a host computer that uses w-bit wide data words, each signal is associated with a data word of which w-1 bits are allocated for w-1 faulty circuits, and the remaining bit is reserved for the fault-free circuit. This way, w-1 faulty circuits and one fault-free circuit can be processed in parallel by use of bitwise logic operations, which corresponds to a speedup factor of approximately

w-1 compared with serial fault simulation. A fault is detected if its bit value differs from that of the fault-free circuit at any of the outputs.

We will reuse the example from serial fault simulation to illustrate the parallel fault simulation process. Assuming that the width of a computer word is three bits, the first bit stores the fault-free (FF) circuit response, and the second and third bits store the faulty responses in the presence of faults f and g, respectively. The simulation results are shown in Table 14.4. Because the fault f, A stuck-at 1, uses the second bit, it is injected by forcing the second bit of the data word of signal A to 1 during fault simulation (shown in the "A_f" column with the forced value underlined—the "A" column corresponds to the fault-free case). Similarly, the "J_g" column depicts how fault g is injected by forcing the third bit to 0.

As we have mentioned, parallel fault simulation is performed by use of bitwise logic operations. For example, the logic value of signal H is obtained by a bitwise AND operation on the data words of signals A and L. (A, J, and L are circled in Table 14.4.) The faulty response of the first pattern is {1, **0**, 1}. This means that fault f is detected (the second bit), but fault g (the third bit) is not. Similarly, the outputs of P_2 and P_3 are {1 **0** 1} and {0 0 **1**}, respectively. In this example, three simulations (in one simulation pass) are performed. Compared with serial fault simulation, which requires nine simulations, parallel fault simulation saves two thirds of the simulation time.

To perform parallel fault simulation with regular parallel logic simulators, one may inject the faults by adding extra logic gates. Figure 14.12 shows how this is done for faults f and g in N. To inject f, a stuck-at one fault, an OR gate (G_f) is inserted, and to force the second bit of A_f to be one without affecting the other two bits, the side input of G_f is set to be 010. Note that the injection of fault f does not affect the fault-free circuit and the faulty circuit with fault g.

Table 14.4 Parallel fault simulation for Figure 14.10

Pat #	Input					Internal					Output	
		A	*Af*	*B*	*C*	*E*	*F*	*L*	*J*	*Jg*	*H*	*K*
P_1	FF	0	0	1	0	1	1	1	0	0	0	1
	f	**0**	**1**	1	0	1	1	1	0	0	**1**	**0**
	g	0	0	1	0	1	1	1	0	0	0	1
P_2	FF	0	0	0	1	1	1	1	0	0	0	1
	f	0	**1**	0	1	1	1	1	0	0	**1**	**0**
	g	0	0	0	1	1	1	1	0	0	0	1
P_3	FF	1	1	0	0	0	0	0	1	1	0	0
	f	1	1	0	0	0	0	0	1	1	0	0
	g	1	1	0	0	0	0	0	1	**0**	0	**1**

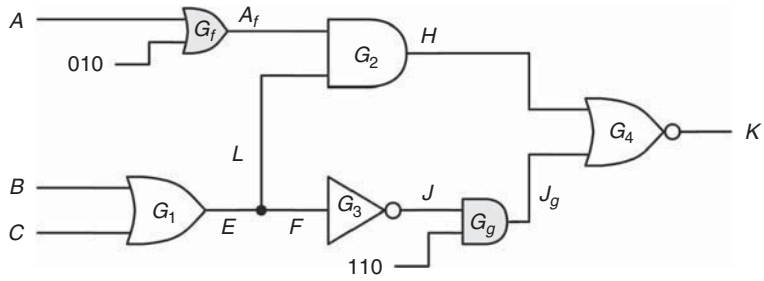


FIGURE 14.12

Fault injection for parallel fault simulation.

Similarly, injecting fault g, a stuck-at 0 fault, is achieved by adding the AND gate G_g and setting its side input to be 110.

Note that the parallel fault simulation technique is applicable to the unit or zero delay models only. More complicated delay models cannot be modeled, because several faults are evaluated at the same time. Furthermore, a simulation pass cannot terminate unless all the faults in this pass are detected. For instance, we cannot drop fault f alone after simulating pattern P_1, because fault g is not detected yet. Parallel fault simulation is best used for simulating the beginning of the test pattern sequence, when a large number of faults are detected by each pattern.

14.3.2.2 *Parallel pattern fault simulation*

Bitwise parallelism can be used to simulate test patterns in parallel. For a host computer with a w-bit data width, the signal values for a sequence of w test patterns are packed into a data word. For the fault-free or faulty circuit, w test patterns can be simulated in parallel by use of bitwise logic operations. This approach was first reported in [Waicukauski 1985], in which it is called ***parallel pattern single fault propagation*** (PPSFP), because one fault at a time is simulated. This approach is especially useful for combinational circuits or full-scan sequential circuits.

In PPSFP, logic simulations on the fault-free circuit are first performed on the first w test patterns, and the circuit outputs are recorded. Then, the faults are simulated one at a time on these w test patterns. For each fault, the simulation results are compared with the correct responses to determine whether the fault is detected. Simulation continues until the fault is detected and dropped, or all the test patterns are simulated. The faulty circuit is restored to its original state, and the next fault is processed. The same procedure repeats until all faults in the fault list are simulated.

The PPSFP results of the fault simulation example are shown in Table 14.5. The "Fault-Free" row lists the fault-free simulation results. Note that the three patterns are packed into one single word and thus are evaluated simultaneously

Table 14.5 PPSFP for Figure 14.10

		Input			Internal					Output
		A	*B*	*C*	*E*	*F*	*L*	*J*	*H*	*K*
Fault Free	P_1	0	1	0	1	1	1	0	0	1
	P_2	0	0	1	1	1	1	0	0	1
	P_3	1	0	0	0	0	0	1	0	0
f	P_1	**1**	1	0	1	1	1	0	**1**	**0**
	P_2	**1**	0	1	1	1	1	0	**1**	**0**
	P_3	1	0	0	0	0	0	1	0	0
g	P_1	0	1	0	1	1	1	0	0	1
	P_2	0	0	1	1	1	1	0	0	1
	P_3	1	0	0	0	0	0	**0**	0	**1**

by use of bitwise logic operations. The "*f*" row represents the simulation results with fault *f* injected. In PPSFP, faults are injected by activating rising or falling events, depending on the stuck-at value, at the faulty signal. Thus, fault *f*, *A* stuck-at one, is injected by activating two rising events on input *A*. The faulty responses are {**0**, **0**, 0}, which indicates that fault *f* is detected by the first and second patterns but not the third one. After fault *f* is simulated, fault *f* is removed by activating two falling events on input *A* at pattern P_1 and P_2. Then, fault *g* is injected by activating one falling event on signal *J* at pattern P_3. A total of three simulation runs are carried out.

Figure 14.13 illustrates the simplified PPSFP flow. Again, fault collapsing is first executed to obtain the collapsed fault list *F*. Then, the first *w* patterns are simulated on the fault-free circuit in parallel, and the good outputs (O_g) are stored. Then, each fault *f* in the fault list *F* is simulated one by one with the same *w* test patterns. A fault is dropped and not simulated against the remaining test patterns if its output response O_f is different from O_g. To fault simulate the next fault, the fault effect of the current fault is removed, and the next fault is injected. This process continues until all faults are either detected or simulated against all test patterns. If the number of test patterns is not an even multiple of the machine word width, only part of the machine word is used when simulating this last batch of patterns.

PPSFP is best suited for simulation of test patterns that come later in the test sequence, where the fault drop rate per pattern is lower. Parallel fault simulation does not work well in this situation, because it cannot terminate a simulation pass until all *w*-1 faults being processed are detected. PPSFP is not suitable for sequential circuits, because the circuit state for test pattern *i* in the *w*-bit

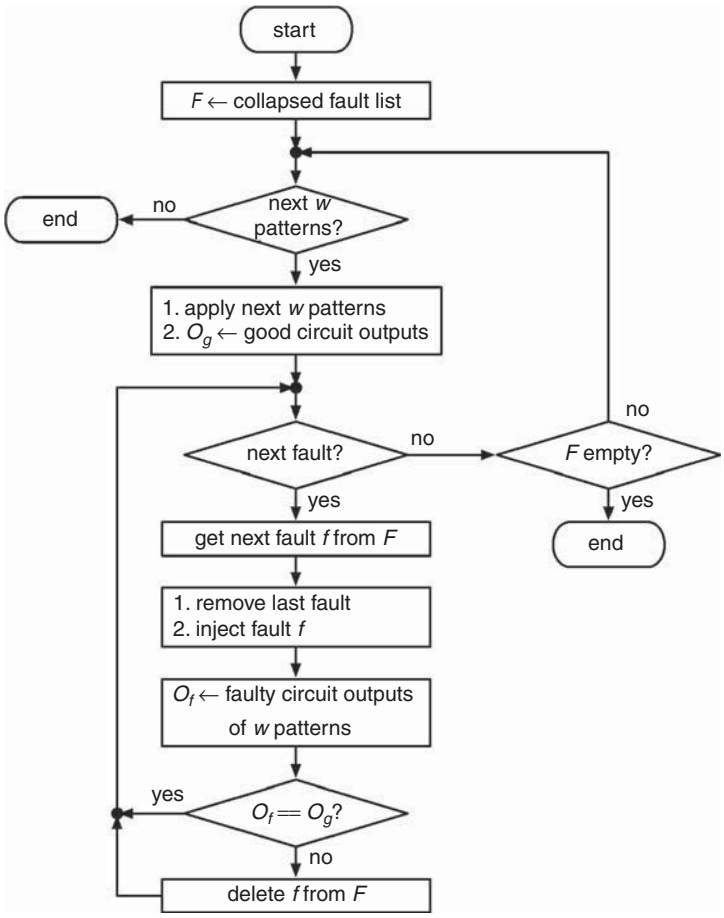


FIGURE 14.13

The PPSFP flowchart.

word depends on the previous i-1 patterns in the word, and this state is not available when the patterns are processed in parallel.

14.3.3 Concurrent fault simulation

Because a fault only affects the logic in the fanout cone from the fault site, the good circuit and faulty circuits typically only differ in a small region. **Concurrent fault simulation** exploits this fact and simulates only the differential parts of the whole circuit [Ulrich 1974]. Concurrent fault simulation is essentially an event-driven simulation with the fault-free circuit and faulty circuits simulated altogether.

In concurrent fault simulation, every gate has a **concurrent fault list**, which consists of a set of **bad gates**. A bad gate of gate x represents an

imaginary copy of gate x in the presence of a fault. Every bad gate contains a fault index and the associated gate I/O values in the presence of the corresponding fault. Initially, the concurrent fault list of gate x contains **local faults** of gate x. The local faults of gate x are faults on the inputs or outputs of gate x. As the simulation proceeds, the concurrent fault list contains not only local faults but also faults propagated from previous stages (called **fault effects**). Local faults of gate x remain in the concurrent fault list of gate x until they are detected.

Figure 14.14 illustrates the concurrent simulation of the example circuit for test pattern P_1. For clear illustration, we demonstrate three faults in this example: A stuck-at one, C stuck-at zero, and J stuck-at zero faults. The concurrent fault lists with bad gates in grey are drawn beside the good gates. The fault indices are labeled in the middle of bad gates and their associated bad gate I/O values are labeled beside their I/O pins. The fault list of G_1, G_2, and G_3 initially contains their local faults: $C/0$, $A/1$, and $J/0$. When we apply the first pattern, three events occur in the primary inputs: u $\rightarrow$ 0 on A, u $\rightarrow$ 1 on B, and u $\rightarrow$ 0 on C. They are **good events**, because they happen in the good circuit. The output of good gate G_1 changes from unknown to one. In the presence of fault $C/0$, the output of faulty G_1 is the same as that of good G_1. A bad gate is **invisible** if its faulty output is the same as the good output. The bad gates C/0 and J/0 are both invisible so they are not propagated to the subsequent stages.

The output of G_2 changes from unknown to zero. In the presence of fault $A/1$, the faulty output changes from unknown to one. Because the faulty output differs from the good output, bad gate $A/1$ becomes visible. A bad gate is **visible** if its faulty output is different from the good output. The visible bad gate

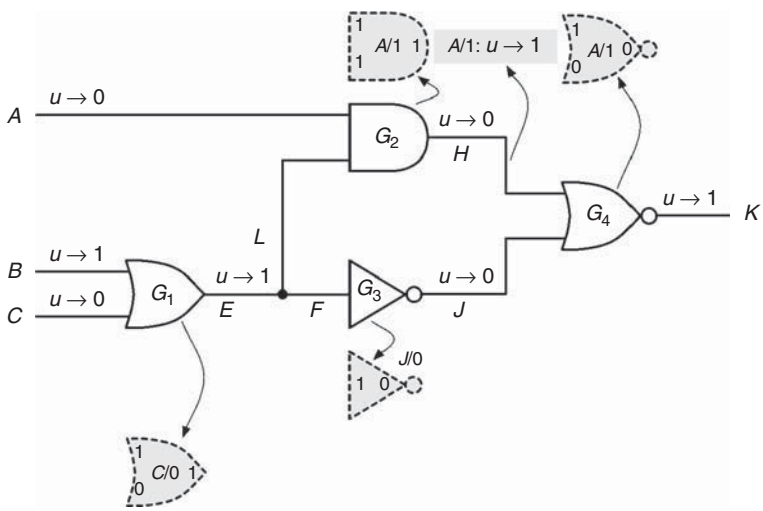


FIGURE 14.14

Concurrent fault simulation (P_1).

$A/1$ creates a bad event u → 1 on net H (in gray). A **bad event** does not occur in the good circuit; it only occurs in the faulty circuit of the corresponding fault. A new copy of bad gate $A/1$ is added to the concurrent fault list of G_4, because it has one input different from the good gate. It is said that bad gate $A/1$ **diverges from** its good gate. Finally, fault $A/1$ is detected because the faulty output K is different from the good output. At this time, we could drop detected fault $A/1$, but we keep it for illustration purposes.

Figure 14.15 illustrates the concurrent fault simulation for test pattern P_2. Two good events occur in this figure: 0 → 1 on C and 1 → 0 on B. The bad gate $C/0$, which was invisible in pattern P_1, now becomes **newly visible**. The newly visible bad gate creates a bad event, net E falls to zero, which in turn creates two divergences in G_2 and G_3. The former is invisible, but the latter creates a bad event, net J rises to one. Finally, the concurrent fault list of G_4 contains two bad gates; both faults $A/1$ and $C/0$ are detected.

For the last test pattern P_3 (Figure 14.16), two good events occur at primary inputs A and C. The bad gate $C/0$ now becomes invisible. The bad gate $C/0$ is deleted from the concurrent fault list of G_3. A bad gate **converges to** its good gate if it is not a local fault and its I/O values are identical to those of the good gate. Similarly, the other bad gates $C/0$ also converge to G_2 and G_4. Note that bad gate $C/0$ does not converge to G_1, because it is a local fault for G_1. The bad gate $A/1$ can be examined in the same way. For gate G_3, although the faulty output of bad gate $J/0$ does not change, the good event 0 → 1 on J makes bad gate $J/0$ newly visible.

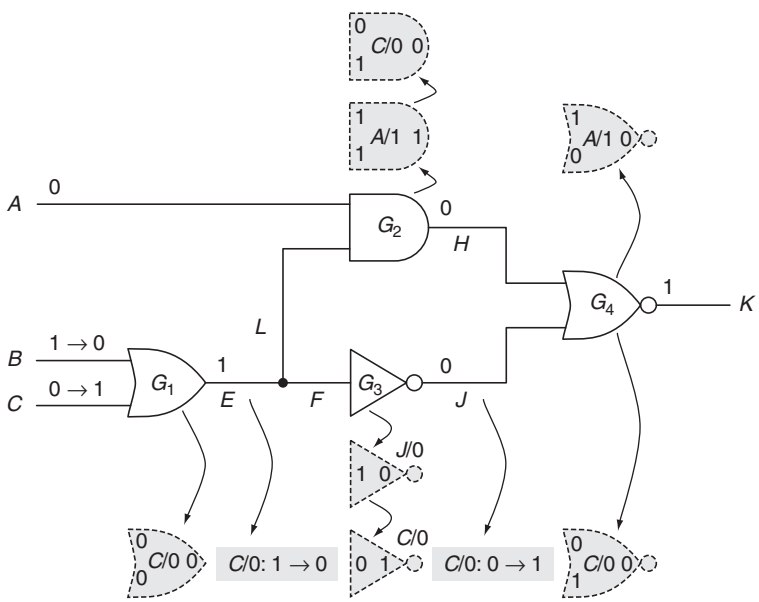


FIGURE 14.15

Concurrent fault simulation (P_2).

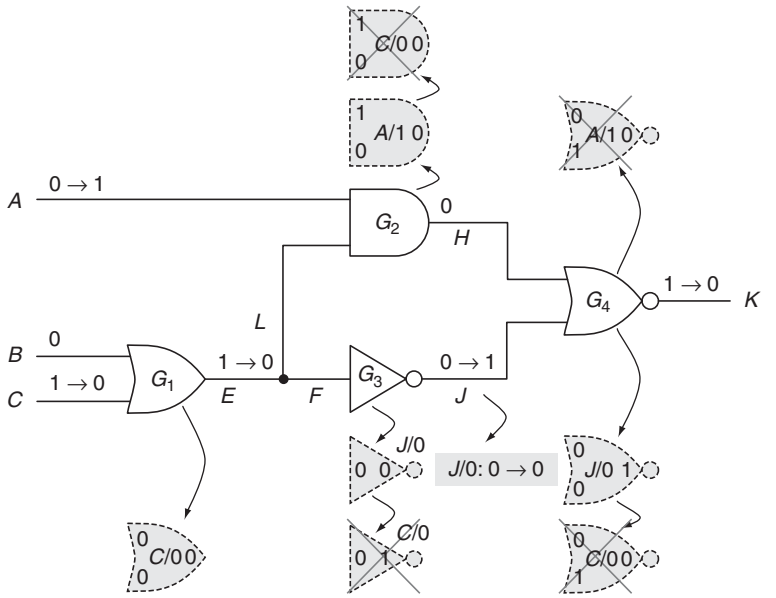


FIGURE 14.16

Concurrent fault simulation (P_3).

The newly visible event (in gray) is propagated to G_4, and a new bad gate $J/0$ diverges from G_4. Eventually, the fault $J/0$ is detected by pattern P_3.

Figure 14.17 shows a simplified concurrent fault simulation flowchart. The fault simulator applies one pattern at a time. The concurrent fault simulation is an event-driven simulation with both good events and bad events simulated at the same time. The events on the gate inputs are first analyzed. A good event affects both good and bad gates but a bad event only affects bad gates of the corresponding fault. After the analysis, events are then executed. The diverged bad gates and converged bad gates are added to or deleted from the fault list, respectively. Determining whether a bad gate diverges or converges depends on three factors: the visibility, the bad event, and the concurrent fault list (see [Abramovici 1994] for more details). After the event execution, new events are computed at the gate outputs. If an event reaches the primary outputs, detected faults can be removed from concurrent fault lists of all gates. This process repeats until there are no more test patterns, or no undetected faults.

14.3.4 **Differential fault simulation**

Concurrent fault simulation constructs the state of the faulty circuit from that of the same faulty circuit of the previous test pattern. Concurrent fault simulation has a potential memory problem, because the size of the concurrent fault list changes at runtime. In contrast, the single fault propagation technique

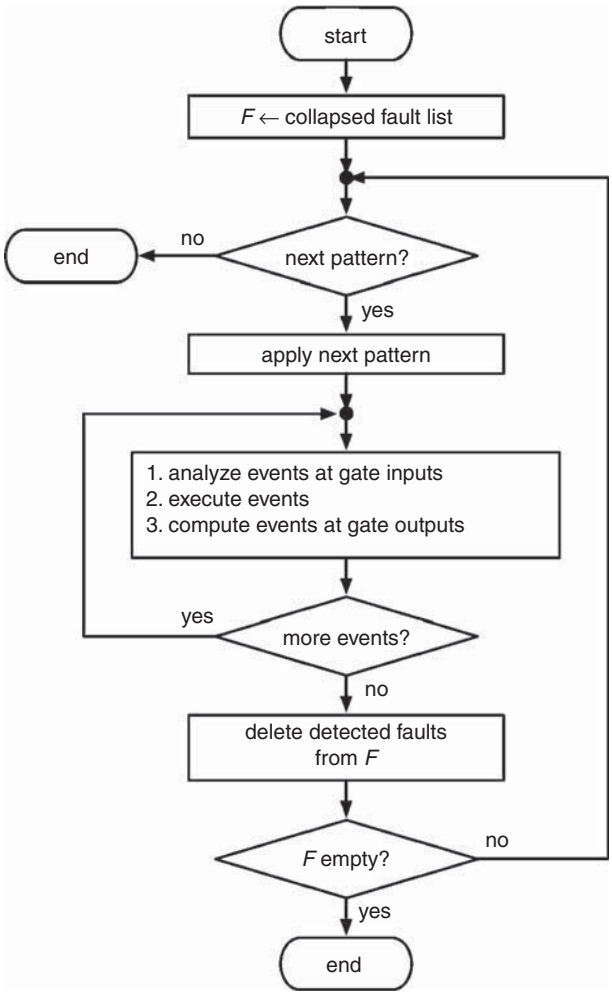


FIGURE 14.17

Concurrent fault simulation flowchart.

constructs the state of the faulty circuit from that of the good circuit. For sequential circuits, the single fault propagation technique would require a large overhead to store the states of the good circuit. Neither of the preceding two techniques are good for sequential fault simulation. Differential fault simulation combines the merits of concurrent fault simulation and single fault propagation techniques [Cheng 1989]. The idea is to simulate, in turn, every faulty circuit by tracking only the difference between a faulty circuit and the last simulated one. An event-driven simulator can easily implement differential fault simulation with the differences injected as events. This differential fault simulation technique

	P_1	P_2	…	P_i	P_{i+1}	…	P_n
Good	G_1	G_2	…	G_i	G_{i+1}	…	G_n
f_1	$F_{1,1}$	$F_{1,2}$	…	$F_{1,i}$	$F_{1,i+1}$	…	$F_{1,n}$
f_2	$F_{2,1}$	$F_{2,2}$	…	$F_{2,i}$	$F_{2,i+1}$	…	$F_{2,n}$
.	.	.	…	.	.	…	.
f_k	$F_{k,1}$	$F_{k,2}$	…	$F_{k,i}$	$F_{k,i+1}$	…	$F_{k,n}$
f_{k+1}	$F_{k+1,1}$	$F_{k+1,2}$	…	$F_{k+1,i}$	$F_{k+1,i+1}$	…	$F_{k+1,n}$
.	.	.	…	.	.	…	.
f_m	$F_{m,1}$	$F_{m,2}$	…	$F_{m,i}$	$F_{m,i+1}$	…	$F_{m,n}$

FIGURE 14.18

Differential fault simulation.

has been further combined with the parallel fault simulation technique, as implemented in **PROOFS** [Niermann 1992].

Figure 14.18 illustrates how differential fault simulation works. First, the first pattern P_1 is simulated on the good circuit G_1, and the good primary outputs are stored. Then the faulty circuit ($F_{1,1}$) is simulated with fault f_1 injected as an event. The first subscript indicates the fault and the second subscript indicates the pattern. The difference of states between G_1 and $F_{1,1}$ is stored. Note that only the states of storage elements, such as flip-flops, are stored, so the memory needed is small compared with concurrent fault simulation. If the primary outputs of $F_{1,1}$ and G_1 are not the same, then fault f_1 is detected. Following F_1 the second faulty circuit ($F_{2,1}$) is simulated with f_1 removed and f_2 injected. Similarly, the difference of states between F_1 and F_2 is stored. The preceding process continues until pattern P_1 has been simulated for all faults (f_1 to f_m).

Following the first pattern, the state of the good circuit G_2 is restored and the second pattern P_2 is applied. After the fault-free simulation, the primary outputs of G_2 are stored. The state of faulty circuit $F_{1,2}$ is restored by injecting the difference of G_1 and $F_{1,1}$. The fault f_1 is again injected as an event. The differential fault simulation for P_2 is the same as that of pattern P_1. Differential fault simulation goes in the direction of the arrows in Figure 14.18—G_i, $F_{1,i}$, $F_{2,i}$, . . . , $F_{m,i}$, G_{i+1}, $F_{1,i+1}$, ...

Figure 14.19 shows a simplified flowchart for differential fault simulation. For every test pattern, a fault-free simulation is performed first. Then the faulty circuits are simulated one after another. The states of every circuit are restored from the last simulation. If the faulty circuit outputs are different from the good outputs, the fault is detected and dropped. The state difference of every circuit is stored. With fault dropping, the state difference of the dropped fault must be accumulated into the state differences of its next undetected fault. This process repeats until there are no test patterns or no undetected faults.

The problem with differential fault simulation is that the order of events caused by fault sites is not the same as the order of the timing of their occurrence. If the circuit behavior depends on the gate delay of the circuit, the timing

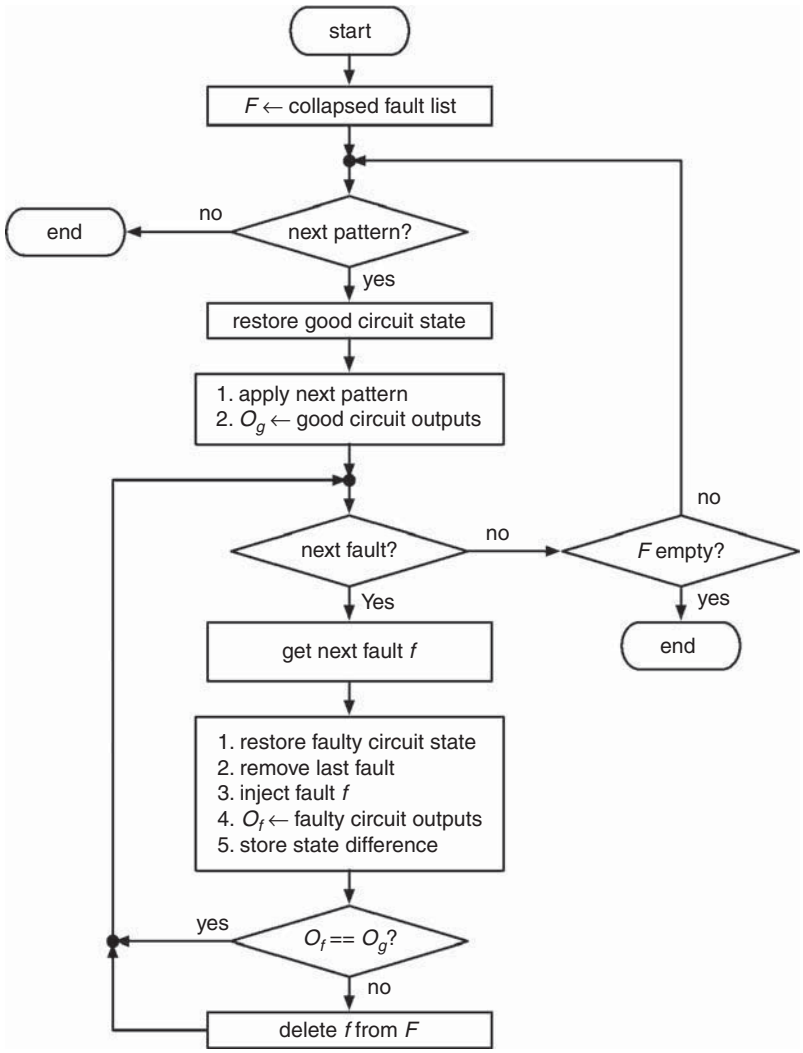


FIGURE 14.19

Differential fault simulation flowchart.

information of every event must be included. This solution, however, may potentially require high memory consumption.

14.3.5 Comparison of fault simulation techniques

In terms of simulation speed, it is apparent that serial fault simulation is the slowest among all the techniques. Differential fault simulation is shown to be up to twelve times faster than concurrent fault simulation and PPSFP [Cheng 1989], when the

sequential circuit under test does not contain memories, such as *static random-access memories* (SRAMs) and *dynamic random-access memories* (DRAMs).

Memory use is, in general, not a problem for serial fault simulation, because it deals with one fault at a time. Similarly, parallel fault simulation and PPSFP do not require much more memory than the fault-free simulation. Concurrent fault simulation has severe memory problems, because the size of the concurrent fault list is unpredictable. Furthermore, the I/O values of every bad gate in the concurrent fault simulation must be recorded. Differential fault simulation relieves the memory management problem of concurrent fault simulation, because only the difference in storage elements is stored.

When the unknown (X) and/or high-impedance (Z) values are present in the circuit, a multiple-valued fault simulation becomes necessary. Serial fault simulation has no problem in handling multiple-valued fault simulation, because it can be realized with a regular logic simulator. In contrast, to exploit bitwise word parallelism, it is more difficult for parallel fault simulation or PPSFP to handle X or Z. In concurrent fault simulation, dealing with multiple-valued simulations is straightforward, because every bad gate is evaluated in the same way as in the fault-free simulation. Finally, differential fault simulation can simulate X or Z without a problem, because it is based on event-driven simulation.

From the aspect of delay and functional modeling capability, serial fault simulation does not encounter any difficulty. Parallel fault simulation and PPSFP cannot take delay or functional models into account, because they pack the information of multiple faults or test patterns into the same word and rely on bitwise logic operations. Being event-driven, both concurrent and differential fault simulation techniques are capable of handling functional models; however, only the former is able to process circuit delays.

When sequential circuits are of concern, serial and parallel fault simulation techniques do not have a problem. The PPSFP technique, however, is not suited for sequential circuit simulation, because a large memory space is required to store the states of the fault-free circuit. Concurrent and differential fault simulations are able to perform sequential fault simulation without difficulty.

On the basis of the previous discussions, PPSFP and parallel fault simulation techniques are currently the most popular fault simulation techniques for combinational (full-scan) circuits. On the other hand, concurrent fault simulation techniques have been widely adopted for sequential circuits embedded with memories, whereas differential fault simulation techniques are mostly suitable for sequential circuits without memories. Algorithm switching has also been used to improve performance. Parallel fault simulation can be used when the fault drop rate per test pattern is high, and then PPSFP is used when more patterns are required to drop each fault.

Even for fault simulation techniques that are efficient in time and memory, the problems of memory explosion and long simulation time still exist as *integrated circuit* (IC) complexity continues growing. To overcome the memory problem, the **multiple-pass fault simulation** approach is often adopted. The idea of

multiple-pass fault simulation is to partition the faults into smaller groups, each of which is simulated independently. If the faults are well partitioned, multiple-pass fault simulation prevents the memory explosion problem. To further reduce the fault simulation time, **distributed fault simulation** approaches may be used. Distributed fault simulation divides the whole fault simulation into smaller tasks, each of which is performed independently on a separate processor.

There are several alternatives to fault simulation. The fault-sampling technique was proposed to simulate only a sampled group of faults [Butler 1974]. Critical path tracing is another alternative to fault simulation [Abramovici 1984]. Instead of performing actual fault simulation, the ***statistical fault analysis*** (STAFAN) approach proposes to use probability theory to estimate the expected value of fault coverage [Jain 1985]. These alternatives to fault simulation have also been extensively discussed in [Abramovici 1994], [Bushnell 2000], and [Wang 2006].

14.4 TEST GENERATION

First, consider the single stuck-at fault model. Figure 14.20 shows a circuit with a single stuck-at fault in which signal *d* is tied to logic 1 (*d*/1). A logic 0 must be applied to node *d* from the primary inputs of the circuit to produce a difference between the fault-free (or good) circuit and the circuit with the stuck-at fault present. Next, to observe the effect of the fault, a logic 1 must be applied to signal *c*. So, if the fault *d*/1 is present, it can be detected at the output *e* with the derived vector. Test generation attempts to generate test vectors for every possible fault in the circuit. In this example, in addition to the *d*/1 fault, faults such as *a*/1, *b*/1, and *e*/1 are also targeted by the test generator. Because some of the faults in the circuit can be logically equivalent, no test can be obtained to distinguish between them. Thus, equivalence fault collapsing as described in Section 14.2 is often used to identify equivalent faults *a priori* to reduce the number of faults that must be targeted [Abramovici 1994; Bushnell 2000; Jha 2003]. Subsequently, the ATPG is only concerned with generating test vectors for each fault in the collapsed fault list.

14.4.1 Random test generation

Random test generation (RTG) is one of the simplest methods for generating vectors. Vectors are randomly generated and fault-simulated (or fault-graded) on the ***circuit under test*** (CUT). Because no specific fault is targeted, the

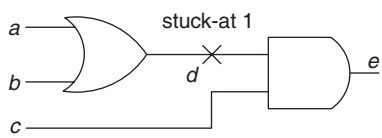


FIGURE 14.20

Example of a single stuck-at fault.

complexity of RTG is low. However, RTG often results in generating a large number of tests that achieves sub-par fault coverage because of the difficult-to-test faults.

In RTG, logic values are randomly generated at the primary inputs, with equal probability of assigning a logic 1 or logic 0 to each primary input. Thus, the random vectors are uniformly distributed in the test set. Note that the random test set is not truly random, because a pseudo-random number generator is generally used. In other words, the random test set can be repeated with the same pseudo-random number generator. Nevertheless, the vectors generated hold the necessary statistical properties of a random vector set.

The **level of confidence** one can have on a random test set T can be measured as the probability that T can detect all the stuck-at faults in the circuit. For N random vectors, the **test quality** t_N indicates the probability that all detectable stuck-at faults are detected by these N random vectors. Thus, the test quality of a random test set highly depends on the circuit under test. Consider a circuit with an eight-input AND gate (or equivalently a cone of seven two-input AND gates) illustrated in Figure 14.21. Although achieving a logic 0 at the output of the AND gate is easy, getting a logic 1 is difficult. A logic 1 would require all the inputs to be at logic 1. If the RTG assigns each primary input with an equal probability of logic 0 or logic 1, the chance of getting eight logic 1's simultaneously would only be $0.5^8 = 0.0039$. In other words, the AND gate output stuck-at-0 fault would be difficult to test by the RTG. Such faults are called **random-pattern resistant faults**.

As discussed earlier, the quality of a random test set depends on the underlying circuit. More random-pattern resistant faults will more likely reduce the quality of the random test set. To tackle the problem of targeting random-pattern resistant faults, biasing is required so the input vectors are no longer viewed as uniformly distributed. Consider the same eight-input AND gate example again. If each input of the AND gate has a much higher probability of

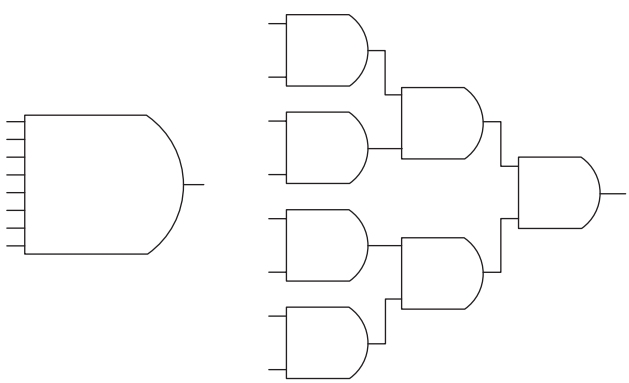


FIGURE 14.21

Two equivalent circuits.

receiving a logic 1, the probability of getting a logic 1 at the output of the AND gate significantly increases. For example, if each input has a 75% probability of receiving a logic 1, then getting a logic 1 at the output of the AND gate now becomes $0.75^8 = 0.1001$, rather than the previous 0.0039.

Determining the optimal bias values for each primary input that can achieve the highest coverage is not an easy task. Thus, rather than trying to obtain the best set of values, the objective is frequently to increase the probabilities for those difficult-to-control and difficult-to-observe nodes in the circuit. For instance, suppose a circuit has an eight-input AND gate; any fault that requires the AND gate output equal to logic 1 for detection will be considered difficult to test. It would then be beneficial to attempt to increase the probability of obtaining a logic 1 at the output of this AND gate.

Another issue regarding random test generation is the number of random vectors needed. Given a combinational circuit with n primary inputs, there are clearly 2^n possible input vectors. One can express the probability of detecting fault f by any random vector to be:

$$d_f = T_f / 2^n$$

where T_f is the set of vectors that can detect fault f. Consequently, the probability that a random vector will not detect f (*i.e.*, f escapes a random vector) is: $e_f = 1 - d_f$.

Therefore, given N random vectors, the probability that none of the N vectors detect fault f is:

$$e_f^N = (1 - d_f)^N$$

In other words, the probability that at least one of N vectors will detect fault f is:

$$1 - (1 - d_f)^N$$

If the detection probability, d_f, for the hardest fault is known, N can be readily computed by solving the following inequality:

$$1 - (1 - d_f)^N \geq p$$

where p is the probability that N vectors should detect fault f.

If the detection probability is not known, it can be computed directly from the circuit. The detection probability of a fault is directly related to: (1) the controllability of the line that the fault is on and (2) the observability of the fault-effect to a primary output. The controllability and observability computations have been introduced previously in the chapter on design for testability. It is worth noting that the minimum detection probability of a detectable fault f can be determined by the output cone in which f resides. In fact, if f is detectable, it must be excited and propagated to at least one primary output, as illustrated in Figure 14.22. It is clear that all the primary inputs necessary to excite f and propagate the fault-effect must reside in the cone of the output

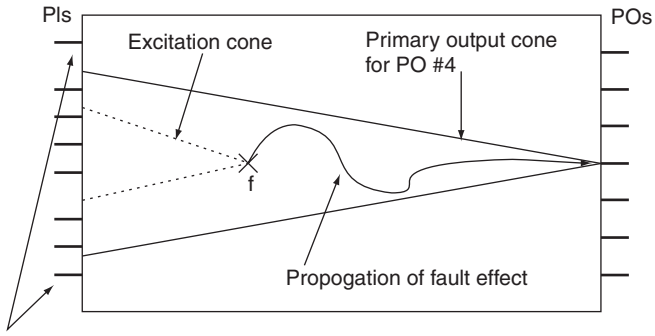


FIGURE 14.22

Detection of a fault.

to which f is detected. Thus, the detection probability for f is at least $(0.5)^m$, where m is the number of primary inputs in the cone of the corresponding primary output. Taking this concept a step further, the detection probability of the most difficult fault can be obtained with the following lemma [David 1976; Shedletsky 1977].

Lemma 1: In a combinational circuit with multiple outputs, let n_{max} be the number of primary inputs that can lead to a primary output. Then, the detection probability for the most difficult detectable fault, d_{min}, is:

$$d_{\min} \geq (0.5)^{nmax}$$

Proof

The proof follows from the preceding discussion.

14.4.1.1 *Exhaustive testing*

If the combinational circuit has few primary inputs, **exhaustive testing** may be a viable option, where every possible input vector is enumerated. This may be superior to random test generation, because RTG can produce duplicated vectors and may miss certain ones.

In circuits in which the number of primary inputs is large, exhaustive testing becomes prohibitive. However, on the basis of the results of Lemma 1, it may be possible to partition the circuit and only exhaust the input vectors within each cone for each primary output. This is called **pseudo-exhaustive testing**. In doing so, the number of input vectors can be drastically reduced. When enumerating the input vectors for a given primary output cone, the values for the primary inputs that are outside the cone are simply assigned random values. Therefore, if a circuit has three primary outputs, each has a corresponding primary output cone. Note that these three primary output cones may overlap. Let $n1$, $n2$, and $n3$ be the number of primary inputs corresponding to these three cones. Then the number of pseudo-exhaustive vectors is simply at most $2^{n1} + 2^{n2} + 2^{n3}$.

14.4.2 Theoretical Background: Boolean difference

Consider the circuit shown in Figure 14.23. Let the target fault be the stuck-at-0 fault on primary input y. Recall the high-level concept of test generation illustrated in Figure 14.1, where the objective is to distinguish the fault-free circuit from the faulty circuit. In the example circuit shown in Figure 14.23, the faulty circuit is the circuit with y stuck at 0. Note that the circuit output can be expressed as a Boolean formula:

$$f = xy + y'z$$

Let $f2$ be the faulty circuit with the fault y/0 present. In other words,

$$f2 = f(y = 0)$$

To distinguish the faulty circuit $f2$ from the fault-free counterpart f, any input vector that can make $f \oplus f2 = 1$ would suffice. Furthermore, because the aim is test generation, the target fault must be excited. In this example, the logic value on primary input y must be logic 1 to excite the fault $y/0$. Putting these two conditions together, the following equation is obtained:

$$y \cdot f(y = 1) \oplus f(y = 0) = 1 \tag{14.1}$$

Note that $f(y = 1) \oplus f(y = 0)$ indicates the exclusive-or operation on the two functions $f(y = 1)$ and $f(y = 0)$; it evaluates to logic 1 if and only if the two functions evaluate to opposing values. In terms of ATPG, this is synonymous to propagating the fault effect at node y to the primary output f. Therefore, any input vector on primary inputs x, y, and z that can satisfy Equation (14.1) is a valid test vector for fault $y/0$:

$$y \cdot f(y = 1) \oplus f(y = 0) = y(x \oplus z) = y(xz' + x'z) = xyz' + x'yz$$

In this running example, the two vectors $xyz = \{110, 011\}$ are candidate test vectors for fault $y/0$. Formally, $f(y = 1) \oplus f(y = 0)$ is called the **Boolean difference** of f with respect to y and is often written as:

$$df/dy = f(y = 1) \oplus f(y = 0)$$

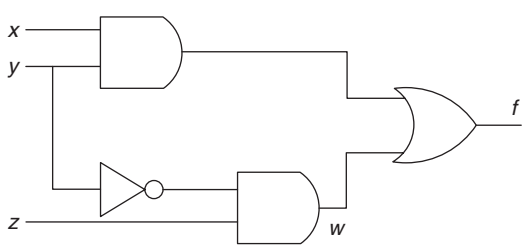


FIGURE 14.23

Example circuit to illustrate the concept of Boolean difference.

In general, if f is a function of x_1, x_2, ... , x_n, then:

$$\mathrm{d}f/\mathrm{d}x_i = f(x_1, x_2, \ldots, x_i = 1, \ldots, x_n) \oplus f(x_1, x_2, \ldots, x_i = 0, \ldots, x_n)$$

In terms of test generation, for any target fault on some fault α/v, the set of all vectors that can *propagate* the fault-effect to the primary output f is then those vectors that can satisfy:

$$\mathrm{d}f/\mathrm{d}\alpha = 1$$

(Note that this is independent of the polarity of the fault, whether it is stuck-at-0 or stuck-at-1.) Next, the constraint that the fault must be excited, α set to value v', must be added. Subsequently, the set of test vectors that can detect the fault becomes all those input values that can satisfy the following equation:

$$(\alpha = v') \cdot \mathrm{d}f/\mathrm{d}\alpha = 1 \tag{14.2}$$

Consider the same circuit shown in Figure 14.23 again. Suppose the target fault is $w/0$. The same analysis can be performed for this new fault. The set of test vectors that can detect $w/0$ is simply:

$$\begin{aligned}
& w \cdot \mathrm{d}f/\mathrm{d}w = 1 \\
\Rightarrow & w \cdot f(w = 1) \oplus f(w = 0) = 1 \\
\Rightarrow & w \cdot (1 \oplus xy) = 1 \\
\Rightarrow & w \cdot (xy)' = 1 \\
\Rightarrow & w \cdot (x' + y') = 1 \\
\Rightarrow & wx' + wy' = 1
\end{aligned}$$

Now, w can be expanded from the circuit shown in the figure to be $w = y' \cdot z$. Plugging this into the equation above gives us:

$$\begin{aligned}
& w \cdot x' + w \cdot y' = 1 \\
\Rightarrow & \mathrm{y}' \cdot \mathrm{z}x' + \mathrm{y}' \cdot \mathrm{z} \cdot \mathrm{y} = 1 \\
\Rightarrow & \mathrm{x}' \cdot \mathrm{y}'\mathrm{z} + \mathrm{y}' \cdot \mathrm{z} = 1 \\
\Rightarrow & \mathrm{y}' \cdot \mathrm{z} = 1
\end{aligned}$$

Therefore, the set of vectors that can detect $w/0$ is {001, 101}.

14.4.2.1 *Untestable faults*

If the target fault is untestable, it would be impossible to satisfy Equation (14.2). Consider the circuit shown in Figure 14.24. Suppose the target fault is $z/0$. Then the set of vectors that can detect z/0 are those that can satisfy:

$$\begin{aligned}
& z \cdot \mathrm{d}f/\mathrm{d}z = 1 \\
\Rightarrow \quad & z \cdot f(z = 1) \oplus f(z = 0) = 1 \\
\Rightarrow \quad & z \cdot (xy \oplus xy) = 1 \\
\Rightarrow \quad & z \cdot 0 = 1 \\
\Rightarrow \quad & \text{UNSATISFIABLE}
\end{aligned}$$

In other words, there exists no input vectors that can satisfy $z \cdot \mathrm{d}f/\mathrm{d}z = 1$, indicating that the fault $z/0$ is untestable.

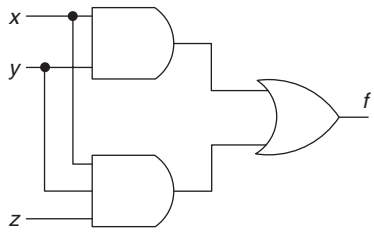


FIGURE 14.24

Example circuit for an untestable fault.

14.4.3 Designing a stuck-at ATPG for combinational circuits

In deterministic ATPG algorithms, there are two main tasks. The first is to excite the target fault, and the second is to propagate the fault-effect to a primary output. Because the logic values in both the fault-free and faulty circuits are needed, composite logic values are used. For each signal in the circuit, the values v/v_f are needed, where v denotes the value for the signal in the fault-free circuit, and v_f represents the value in the corresponding faulty circuit. Whenever $v = v_f$, v is sufficient to denote the signal value. To facilitate the manipulation of such composite values, a 5-valued algebra was proposed [Roth 1966], in which the five values are 0, 1, X, D, and $\bar{D}$; 0, 1, and X are the conventional values found in logic design for true, false, and "don't care." D represents the composite logic value 1/0 and $\bar{D}$ represents 0/1. Boolean operators such as AND, OR, NOT, and XOR can work on the 5-valued algebra as well. The simplest way to perform Boolean operations is to represent each composite value into the v/v_f form and operate on the fault-free value first, followed by the faulty value. For example, 1 AND D is 1/1 AND 1/0. AND-ing the fault-free values yields 1 AND 1 = 1, and AND-ing the faulty values yields 1 AND 0 = 0. So the result of the AND operation is 1/0 = D. As another example,

$$\begin{aligned} D \text{ OR } \bar{D} &= 1/0 \text{ OR } 0/1 \\ &= 1/1 \\ &= 1 \end{aligned}$$

Tables 14.6, 14.7, and 14.8 show the AND, OR, and NOT operations for the 5-valued algebra, respectively. Operations on other Boolean conjunctives can be constructed in a similar manner.

14.4.3.1 *A naive ATPG algorithm*

A very simple and naive ATPG algorithm is shown in Algorithm 14.3, in which combinational circuits with fanout structures can be handled.

Table 14.6 AND Operation

AND	**0**	**1**	***D***	$\overline{D}$	***X***
0	0	0	0	0	0
1	0	1	D	$\overline{D}$	*X*
D	0	*D*	*D*	0	*X*
$\overline{D}$	$\overline{D}$	1	1	$\overline{D}$	*X*
X	X	1	X	X	*X*

Table 14.7 OR Operation

OR	**0**	**1**	***D***	$\overline{D}$	***X***
0	0	1	*D*	$\overline{D}$	*X*
1	1	1	1	1	1
D	*D*	1	*D*	1	*X*
$\overline{D}$	$\overline{D}$	1	1	$\overline{D}$	*X*
X	*X*	1	X	*X*	*X*

Table 14.8 NOT Operation

NOT	
0	1
1	0
D	$\overline{D}$
$\overline{D}$	*D*
X	*X*

Algorithm 14.3 Naive ATPG (*C*, *f*)

1. **while** a fault-effect of *f* has not propagated to a PO and all possible vector combinations have not been tried **do**
2. pick a vector, *v*, that has not been tried;
3. fault simulate *v* on the circuit *C* with fault *f*;
4. **end while**

Note that in an ATPG, the worst-case computational complexity is exponential, because all possible input patterns may have to be tried before a vector is found or that the fault is determined to be undetectable. One may go about line #2 of the algorithm in an intelligent fashion, so a vector is not simply selected indiscriminately. Whether or not intelligence is incorporated, some mechanism is needed to account for those attempted input vectors so no vector would be repeated. If it is possible to deduce some knowledge during the search for the input vector, the ATPG may be able to mark a set of solutions as tried and thus reduce the remaining search space. For instance, after attempting a number of input vectors, this naive ATPG realizes that any input vector with the first primary input set to logic 0 cannot possibly detect the target fault, and it can safely mark all vectors with the first primary input equal to 0 as a tried input vector. Subsequently, only those vectors with the first primary input set to 1 will be selected.

In certain cases, it may not be possible for the ATPG to deduce that all vectors with a given primary input set to some logic value would definitely not qualify to be solution vectors. However, it may be able to make an intelligent guess that input vectors with primary input *#i* set to some specific logic value are more likely to lead to a solution. In such a case, the ATPG would make a **decision** on primary input *#i*. Because the decision may actually be wrong, the ATPG may eventually have to alter its decision, trying the vectors that have the opposite Boolean value on primary input *#i*.

The process of making decisions and reversing decisions will result in a **decision tree**. Each node in the decision tree represents a decision variable. If only two choices are possible for each decision variable, then the decision tree is a binary tree. However, there may be cases in which multiple choices are possible in a general search tree.

Figure 14.25 shows an example decision tree. Although this figure only allows decisions to be made at the primary inputs, in general, this may not be

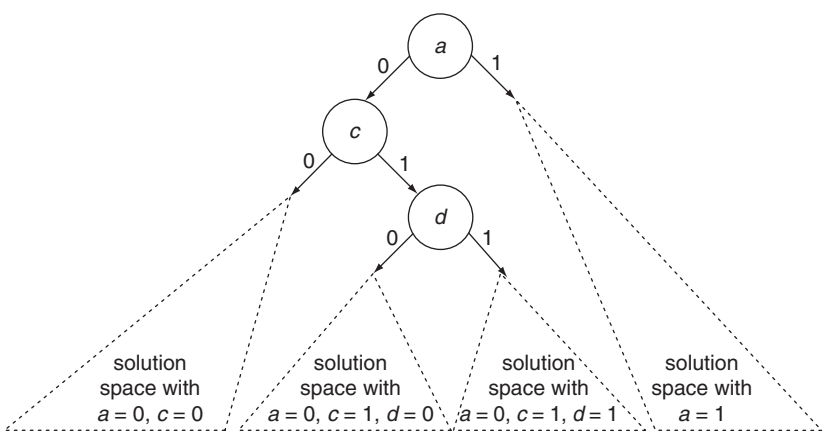


FIGURE 14.25

An example decision tree.

the case. This is used simply to allow the reader to have a clearer picture of the concept behind decision trees. At each decision, the search space is halved. For example, if the circuit has n primary inputs, then there are a total of 2^n possible vectors in the solution space. After a decision is made, the solution spaces under the two branches of a decision node are disjoint. For instance, the space under the decision $a = 1$ does not contain any vectors with $a = 0$. Note that the decision tree for a solution vector may not require the ATPG to *exhaustively* enumerate every possible vector; rather, it *implicitly* enumerates the vectors. If a solution vector exists, there must be a path along the decision tree that leads to the solution. On the other hand, if the fault is undetectable, every path in the decision tree would lead to no solution. It is important to note that a fault may be detected without having made all decisions. For example, the circuit nodes that do not play a role in exciting or propagating the fault would not have to be included in the decision process. Likewise, it may not require all decision variables before the ATPG can determine that it is on the wrong path. For example, if a certain path already sets a value on the fault site such that the fault is not excited, then no value combination on the remaining decision variables can help to excite and propagate the fault. With Figure 14.25 as an example again, suppose the path $a = 0$, $c = 1$, $d = 1$ cannot excite the target fault α. Then, the rest of the decision variables, $b, e, f, \ldots$, cannot undo the effect rendered by $a = 0, c = 1, d = 1$.

14.4.3.1.1 Backtracking

Whenever a conflict is encountered (*i.e.*, a path segment in the decision tree leading to no solution), the search must not continue searching beneath that path but must go back to some earlier point and re-decide on a previous decision. If only two choices are possible for a decision variable, then some previous decision needs to be reversed if the other branch has not been explored before. This reversal of decision is called a **backtrack**. To keep track of where the search spaces have been explored and avoid repeating the search in the same

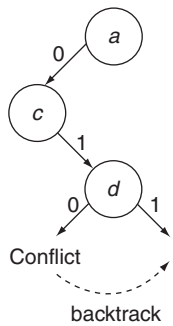


FIGURE 14.26

Backtrack on a decision.

spaces, the easiest mechanism is to reverse the most recent decision made. When reversing any decision, the signal values implied by the assignment of the previous decision variable must be undone.

Consider the decision tree illustrated in Figure 14.26 as an example. Suppose the current decisions made so far are $a = 0$, $c = 1$, $d = 0$, and this causes a conflict in detecting the target fault. Then, the search must reverse the most recently made decision, which is $d = 0$. When reversing $d = 0$ to $d = 1$, all values that resulted from $d = 0$ must be first undone. Then, the search continues with the path $a = 0$, $c = 1$, $d = 1$. If the reversal of a decision also caused a conflict (in this case, reversing $d = 0$ also caused a conflict), then it means $a = 0$, $c = 1$ actually cannot lead to any solution vector that can detect the target fault. The backtracking mechanism would then take the search to the previous decision and attempt to reverse that decision. In the running example, it would undo the decision on d, assigning d to "don't care," followed by reversing of the decision $c = 1$ and searching the portion of the search space under $a = 0$, $c = 0$. Finally, if there is no previous decision that can be reversed, the ATPG concludes that the target fault is undetectable.

Technically, whenever a decision is reversed, say $d = 0$ is reversed to $d = 1$ as shown in Figure 14.26, $d = 1$ is no longer a decision; rather, it becomes an implied value by a subset of the previous decisions made. The exact subset of decisions that implied $d = 1$ can be computed by a **conflict analysis** [Marques-Silva 1999b]. However, the details of conflict analysis are beyond the scope of this chapter and are thus omitted. The reader can refer to [Marques-Silva 1999b] for details of this mechanism. In addition, intelligent conflict analysis can also allow for **nonchronological backtracking**.

14.4.3.2 *A basic ATPG algorithm*

Given a target fault g/v in a fanout-free combinational circuit C, a simple procedure to generate a vector for the fault is shown in Algorithm 14.4, where JustifyFanoutFree() and PropagateFanoutFree() are both recursive functions.

Algorithm 14.4 Basic Fanout Free ATPG (C, g/v)

1. initialize circuit by setting all values to X;
2. JustifyFanoutFree(C, g, v'); /* excite the fault by justifying line g to v' */
3. PropagateFanoutFree(C, g); /* propagate fault-effect from g to a PO */

The JustifyFanoutFree(g, v) function recursively justifies the predecessor signals of g until all signals that need to be justified are, indeed, justified from the primary inputs. The simple outline of the JustifyFanoutFree routine is listed in Algorithm 14.5. In line #10 of the algorithm, controllability measures can be used to select the best input to justify. Selecting a good gate input may help to reach a primary input sooner.

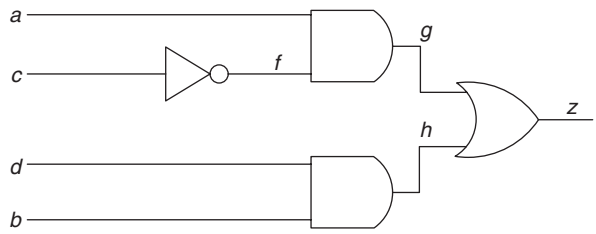


FIGURE 14.27

Example fanout-free circuit.

Consider the circuit C shown in Figure 14.27. Suppose the objective is to justify $g = 1$. According to the preceding algorithm, the following sequence of recursive calls to JustifyFanoutFree would have been made:

call #1: JustifyFanoutFree(C, g, 1)
call #2: JustifyFanoutFree(C, a, 1)
call #3: JustifyFanoutFree(C, f, 1)
call #5: JustifyFanoutFree(C, c, 0)

Algorithm 14.5 JustifyFanoutFree(C, g, v)

```
1. g = v;
2. if gate type of g == primary input then
3.    return;
4. else if gate type of g == AND gate then
5.  if v == 1 then
6.       for all inputs h of g do
7.          JustifyFanoutFree(C, h, 1);
8.       end for
9.  else {v == 0}
10.     h = pick one input of g whose value == X;
11.     JustifyFanoutFree(C, h, 0);
12.   end if
13. else if gate type of g == OR gate then
14. ...
15. end if
```

After these calls to JustifyFanoutFree(), $abcd = 1X0X$ is an input vector that can justify $g = 1$.

Consider another circuit C shown in Figure 14.28. Note that the circuit is not fanout-free, but the preceding algorithm will still work for the objective of

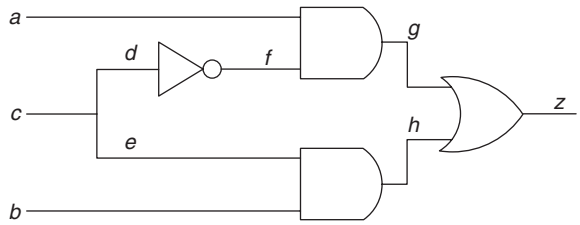


FIGURE 14.28

Example circuit with a fanout structure.

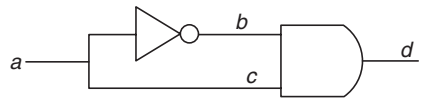


FIGURE 14.29

Circuit with a constant circuit node.

trying to justify the signal $g = 1$. According to the algorithm, the following sequence of calls to the JustifyFanoutFree function would have been made:

call #1: JustifyFanoutFree(C, g, 1)
call #2: JustifyFanoutFree(C, a, 1)
call #3: JustifyFanoutFree(C, f, 1)
call #4: JustifyFanoutFree(C, d, 0)
call #5: JustifyFanoutFree(C, c, 0)

After these five calls to JustifyFanoutFree(), $abc = 1X0$ is an input vector that can justify $g = 1$. Note that in a fanout-free circuit, the JustifyFanoutFree() routine will *always* be able to set g to the desired value v, and no conflict will ever be encountered. However, this is not always true for circuits with fanout structures, such as the circuit shown in Figure 14.29. This is because in circuits with fanout branches, two or more signals that can be traced back to the same fanout stem are **correlated**, and setting arbitrary values on these correlated signals may not always be possible. For example, in the simple circuit shown in Figure 14.29, justifying $d = 1$ is impossible, because it requires both $b = 1$ and $c = 1$, thereby causing a conflict on a.

Consider again the circuit shown in Figure 14.28. Suppose the objective is to set z = 0. On the basis of the JustifyFanoutFree() algorithm, it would first justify both $g = 0$ and $h = 0$. Now, for justifying $g = 0$, suppose it picks the signal f for justifying the objective $g = 0$; it would eventually assign $c = 1$ through the recursive JustifyFanoutFree() function. Next, for justifying $h = 0$, it no longer can choose $e = 0$ as a viable option, because choosing e = 0 will eventually cause a **conflict** on signal c. In other words, a different **decision** has to be made for justifying $h = 0$. In this case, $b = 0$ should be chosen. Although this example is very simple, it illustrates the possibility of making poor decisions, causing potential **backtracks** in the search. In the rest of this chapter, more discussion on avoiding conflicts will be covered.

In the preceding running example, suppose the target fault is $g/0$, and JustifyFanoutFree(C, g, 1) would have successfully excited the fault. With the fault $g/0$ excited, the next step is to propagate the fault-effect to a primary output. Similar to the JustifyFanoutFree() function, PropagateFanoutFree() is a recursive function as well, where the fault-effect is propagated one gate at a time until it reaches a primary output. Algorithm 14.6 illustrates the pseudo-code for one possible implementation of the propagate function.

Again, although the PropagateFanoutFree() routine is meant for fanout-free circuits, it is sufficient for the running example. With the PropagateFanoutFree() function on the fault-effect D at signal g, listed in Algorithm 14.5, the following calls to the JustifyFanoutFree and PropagateFanoutFree functions would have been made:

call #1: PropagateFanoutFree(C, g)
call #2: JustifyFanoutFree(C, b, 0)
call #3: JustifyFanoutFree(C, b, 0)
call #4: PropagateFanoutFree(C, z)

Algorithm 14.6 PropagateFanoutFree(C, g)

```
1.  if g has exactly one fanout then
2.    h = fanout gate of g;
3.    if none of the inputs of h has the value of X then
4.      backtrack;
5.    end if
6.  else {g has more than one fanout}
7.    h = pick one fanout gate of g that is unjustified;
8.  end if
9.  if gate type of h == AND gate then
10.   for all inputs, j, of h, such that j ≠ g do
11.     if the value on j == X then
12.       JustifyFanoutFree(C, j, 1);
13.     end if
14.   end for
15. else if gate type of h == OR gate then
16.   for all inputs, j, of h, such that j ≠ g do
17.     if the value on j == X then
18.       JustifyFanoutFree(C, j, 0);
19.     end if
20.   end for
21. else if gate type of h == ... gate then
22. ...
23. end if
24. PropagateFanoutFree(C, h);
```

Because the fault-effect has successfully propagated to the primary output z, the fault $g/0$ is detected, with the vector $abc = 100$. The reader may notice that once $g/0$ has been excited, it is also propagated to z as well, because $c = 0$ also has made $h = 0$. In other words, the JustifyFanoutFree(C, h, 0) step is unnecessary. However, this is only possible if logic simulation or implication capability is embedded in the BasicFanoutFreeATPG() algorithm. For this discussion, it is not assumed that logic simulation is included.

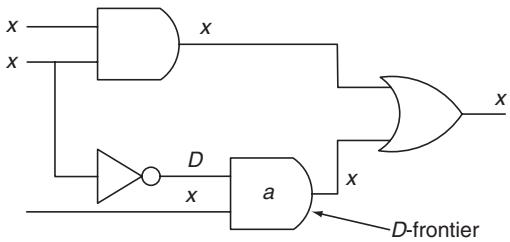
With the same circuit shown in Figure 14.28, consider the fault $g/1$. The BasicFanoutFreeATPG() algorithm will again be used to generate a test vector for this fault. In this case, the ATPG first attempts to justify $g = 0$, followed by propagating the fault-effect to z. During the justification of $g = 0$, the ATPG can pick either a or f as the next signal to justify. At this point, the ATPG must make a **decision**. Testability measures discussed in an earlier chapter can be used as a guide to make more intelligent decisions. In this example, choosing a is considered to be better than f, because choosing a requires no additional decisions to be made. Note that testability measures only serve as a guide to decision selection; they do not guarantee that the guidance will always lead to better decision selection.

It is important to note that in circuits with fanout structures, because the simple JustifyFanoutFree() and PropagateFanoutFree() functions described previously are meant for fanout-free circuits, will not always be applicable as illustrated in some of the earlier examples because of potential conflicts. To generate test vectors for general combinational circuits, there must be mechanisms that will allow the ATPG to avoid conflicts, as well as get out of a conflict when a conflict is encountered. To do so, the corresponding decision tree must be constructed during the search for a solution vector, and backtracks must be enforced for any conflict encountered. The following sections describe a few ATPG algorithms.

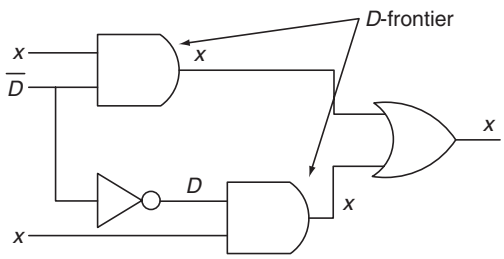
14.4.3.3 *D algorithm*

The D algorithm was proposed to tackle the generation of vectors in general combinational circuits [Roth 1966, 1967]. As indicated by the name of the algorithm, the D algorithm tries to propagate a D or $\overline{D}$ of the target fault to a primary output. Initially, every signal in the circuit has the unknown value, X. At the end of the D algorithm, some signals will be assigned 0, 1, D, or $\overline{D}$, while the rest of the signals may remain as unknown. Note that because each detectable fault can be excited, a fault-effect can always be created. In the following discussion, propagation of the fault-effect will take precedence over the justification of the signals. This allows for enhanced efficiency of the algorithm and for simpler discussion.

Before proceeding to discussing the details of the D algorithm, two important terms should be defined: the ***D*-frontier** and the ***J*-frontier**. The D-frontier consists of all the gates in the circuit whose output value is unspecified and a fault-effect (D or $\overline{D}$) is at one or more of its inputs. For this to occur, one or more inputs of the gate must currently have an unknown value, X. For example, at the start of the D algorithm, for a target fault f there is exactly one D (or $\overline{\mathrm{D}}$) placed in the circuit



(a) *D*-frontier contains one gate



(b) *D*-frontier contains two gates

FIGURE 14.30

Illustrations of *D*-frontier.

corresponding to the stuck-at fault. All other signals currently have a "don't care" value. Thus, the *D*-frontier consists of the successor gate(s) from the line with the fault *f*. Two scenarios of a *D*-frontier are illustrated in Figure 14.30. Clearly, at any time if the *D*-frontier is empty, the fault no longer can be detected. For example, consider Figure 14.30a. If the bottom input of gate *a* is assigned a value of 0, the output of gate *a* will become 0, and the D-frontier now becomes empty. At this time, the search must backtrack and try a different search path.

The *J*-frontier consists of all the gates in the circuit whose output values are known (can be any value in the 5-valued logic except X) but is not justified by its inputs. Figure 14.31 illustrates an example of a *J*-frontier. Thus, to detect the target fault, all gates in the *J*-frontier must be justified; otherwise, some gates in the *J*-frontier must have caused a conflict, where these gates cannot be justified to the desired values.

Having discussed the two fundamental concepts of the *D*-frontier and the *J*-frontier, the explanation for the *D* algorithm can begin. The *D* algorithm begins by trying to propagate the initial *D* (or $\overline{D}$) at the fault site to a primary output. For example, in Figure 14.32, the propagation routine will set all the side inputs of the path necessary (gates $a \rightarrow b \rightarrow c$) to propagate the fault-effect to the respective noncontrolling values. These side input gates, namely x, y, and z, thus form the *J*-frontier, because they are not currently justified. Because the *D* is propagated to the primary output, the *D*-frontier eventually becomes the output gate.

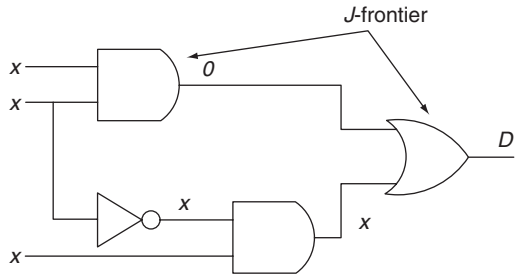


FIGURE 14.31

Illustration of *J*-frontier.

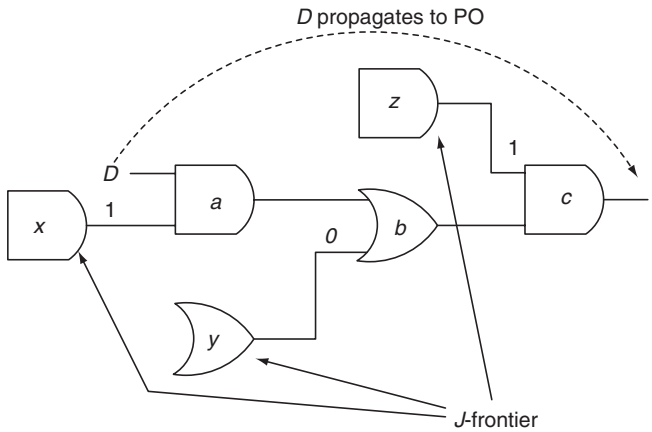


FIGURE 14.32

Propagation of *D*- and *J*-frontier.

Whenever there are paths to choose from in advancing the *D*-frontier, observability values can be used to select the corresponding gates. However, this does not guarantee that the more observable path will definitely lead to a solution. When a D or a $\overline{D}$ has reached a primary output, all the gates in the *J*-frontier must now be justified. This is done by advancing the *J*-frontier backward by placing predecessor gates in the *J*-frontier such that they justify the previous unjustified gates. Similar to propagation of the fault-effect, whenever a conflict occurs, a backtrack must be invoked. In addition, at each step, the *D*-frontier must be checked so the D (or $\overline{D}$) that has reached a primary output is still there. Otherwise, the search returns to the propagation phase and attempts to propagate the fault-effect to a primary output again. The overall procedure for the *D* algorithm is shown in Algorithms 14.7 and 14.8.

Note that the previous procedure has not incorporated any intelligence in the decision-making process. In other words, sometimes it may be possible to determine that some value assignments are not justifiable, given the current

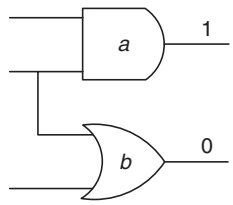


FIGURE 14.33

Conflict in the justification process.

circuit state. For instance, consider the circuit fragment shown in Figure 14.33. Justifying gate $a = 1$ and gate $b = 0$ is not possible, because $a = 1$ requires both of its inputs set to logic 1, whereas $b = 0$ requires both of its inputs set to logic 0. Noting such conflicting scenarios early can help to avoid future backtracks. Such knowledge can be incorporated into line #1 of the *D*-Alg-Recursion() shown in Algorithm 14.8. In particular, implications can be used to identify such potential conflicts, and they are used extensively to enhance the performance of the *D* algorithm (as well as other ATPG algorithms).

Algorithm 14.7 *D*-Algorithm(*C*, *f*)

1. initialize all gates to don't-cares;
2. set a fault-effect (D or $\overline{D}$) on line with fault f and insert it to the *D*-frontier;
3. *J*-frontier = ϕ;
4. result = *D*-Alg-Recursion(*C*);
5. **if** result == success **then**
6. print out values at the primary inputs;
7. **else**
8. print fault *f* is untestable;
9. **end if**

Consider the multiplexer circuit shown in Figure 14.28. If the target fault is *f* stuck-at-0, then, after initializing all gate values to *X*, the D algorithm places a D on line *f*. The algorithm then tries to propagate the fault-effect to *z*. First, it will place $a = 1$ in the *J*-frontier, followed by $b = 0$ in the *J*-frontier. At this time, the fault-effect has reached the primary output. Now, the ATPG tries to justify all unjustified values in the *J*-frontier. Because *a* is a primary input, it is already justified. The other signals in the *J*-frontier are $f = D$ and $h = 0$. For $f = D$, $d = 0$, thereby making $c = 0$. For $b = 0$, either $e = 0$ or $b = 0$ is sufficient. Whichever one it picks, the search process will terminate, as a solution has been found.

Consider the same multiplexer circuit (see Figure 14.28) again. Suppose the target fault now is *f* stuck-at-1. Following the similar discussion as the previous target fault *f*/0, the algorithm initializes the circuit and places a *D* on *f*. Next, to propagate the fault-effect to a primary output, it likewise inserts a = 1 and h = 0 into the *J*-frontier. Now, the ATPG needs to justify all the gates in the J-frontier,

Algorithm 14.8 D-Alg-Recursion(C)

```
1.  if there is a conflict in any assignment or D-frontier is φ then
2.     return failure
3.  end if
4.  /* first propagate the fault-effect to a PO */
5.  if no fault-effect has reached a PO then
6.   while not all gates in D-frontier has been tried do
7.     g = a gate in D-frontier that has not been tried;
8.     set all unassigned inputs of g to non-controlling value and add them
       to the J-frontier;
9.     result = D-Alg-Recursion(C);
10.    if result == success then
11.      return (success);
12.    end if
13.  end while
14. return (failure);
15. end if {fault-effect has reached at least one PO}
16. if J-frontier is φ then
17.    return (success);
18. end if
19. g = a gate in J-frontier;
20. while g has not been justified do
21.    j = an unassigned input of g;
22.    set j = 1 and insert j = 1 to J-frontier;
23.    result = D-Alg-Recursion(C);
24.    if result == success then
25.       return (success);
26.    else try the other assignment
27.       set j = 0;
28.    end if
29. end while
30. return(failure);
```

which includes $a = 1$, $f = D$, and $b = 0$. Because a is a primary output, it is already justified. For $f = D$, $d = 1$. For $b = 0$, suppose it selects $e = 0$. At this time, the J-frontier consists of two gate values: $d = 1$ and $e = 0$. No value assignment on c can satisfy both $d = 1$ and $e = 0$; therefore, a conflict has occurred, and backtrack on the previous decision is needed. The only decision that has been made is $e = 0$ for $b = 0$, because there were two choices possible for

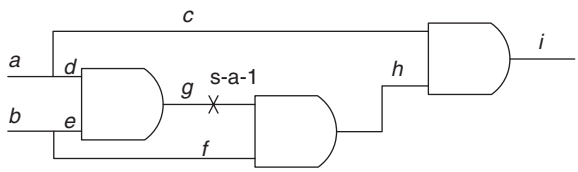


FIGURE 14.34

Example circuit.

justifying $h = 0$. At this time, the value on e is reversed, and $b = 0$ is added to the *J*-frontier. The process continues and all gate values in the *J*-frontier can be successfully justified, ending the process with the vector $abc = 101$.

Note that, in the preceding example, if some learning procedure (such as implications) is present, the decision for $h = 0$ would not result in $e = 0$, because the ATPG would have detected that $e = 0$ would conflict with $d = 1$. This knowledge could potentially improve the performance of the ATPG, which will be discussed later in this chapter.

Consider another example circuit shown in Figure 14.34. Suppose the target fault is $g/1$. After circuit initialization, the *D* algorithm places a $\overline{D}$ on g. Now, the *J*-frontier consists of $g = \overline{D}$ and the *D*-frontier consists of h. To advance the *D*-frontier, f is set to logic 1; $f = 1$ is added to the *J*-frontier, and the *D*-frontier is now i. Next, to propagate the fault-effect to the output, $c = 1$ is added to the *J*-frontier. At this time, the fault-effect has been propagated to the output, and the task is to justify the signal values in the *J*-frontier: $\{g = \overline{D}, f = 1, c = 1\}$. To justify $g = \overline{D}$, two choices are possible: $a = 0$ or $b = 0$. If $a = 0$ is selected, it is necessary to justify $f = 1$, $b = 1$. Finally, $c = 1$ remains in the *J*-frontier which is still unjustified. At this time, a contradiction has occurred ($a = 0$ and $c = 1$), and the search reverses its last decision, changing $a = 0$ to $a = 1$. The search discovers that this reversal also causes a conflict. Thus, a backtrack occurs where line b is chosen instead of a for the previous decision, so a is reset to "don't care." By assigning $b = 0$, a conflict is observed. Reversing b also cannot justify all the *J*-frontier. At this time, backtracking on b leads to no prior decisions. Thus, target fault $g/1$ is declared to be untestable.

14.4.4 PODEM

In the *D* algorithm, the decision space encompasses the entire circuit. In other words, every internal gate could be a decision point. However, noting that the end result of any ATPG algorithm is to derive a solution vector at the primary inputs and that the number of primary inputs generally is much fewer than the total number of gates, it may be possible to arrive at a very different ATPG algorithm that makes decisions only at primary inputs rather than at internal nodes of the circuit.

The ***path-oriented decision-making*** (PODEM) algorithm [Goel 1981] is based on this notion and makes decisions only at the primary inputs. Similar

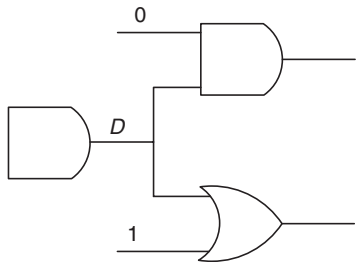


FIGURE 14.35

No *X* path.

to the *D* algorithm, a *D*-frontier is kept. However, because decisions are made at the primary inputs, the *J*-frontier is unnecessary. At each step of the ATPG search process, it checks whether the target fault is excited. If the fault is excited, it then checks whether there is an *X*-path from at least one fault- effect in the *D*-frontier to a primary output, where an *X*-path is a path of unspecified values from the fault-effect to a primary output. If no *X*-path exists, it means that all the fault-effects in the *D*-frontier are blocked, as illustrated in Figure 14.35, where both possible propagation paths of the *D* have been blocked. Otherwise, PODEM will pick the best *X*-path to propagate the fault-effect. Note that if the target fault has not been excited, the first steps of PODEM will be to excite the fault.

The basic flow of PODEM is illustrated in Algorithms 14.9 and 14.10. Although it is still a deterministic search algorithm, the decisions are limited to the primary inputs. All internal signals obtain their logic values by means of logic simulation (or implications) from the decision points. As a result, no conflict will ever occur at the internal signals of the circuit. The only possible conflicts in PODEM are either (1) the target fault is not excited or (2) the *D*-frontier becomes empty. In either of these cases, the search must backtrack.

Algorithm 14.9 PODEM(*C*, *f*)

```
1. initialize all gates to don't-cares;
2. D-frontier = φ;
3. result = PODEM-Recursion(C);
4. if result == success then
5.    print out values at the primary inputs;
6. else
7.    print fault f is untestable;
8. end if
```

Algorithm 14.10 PODEM-Recursion(*C*)

```
1. if fault-effect is observed at a PO then
2.     return (success);
3. end if
4. (g, v) = getObjective(C);
5. (pi, u) = backtrace (g, v);
6. logicSimulate_and_imply (pi, u);
7. result = PODEM-Recursion(C);
8. if result == success then
9.     return(success);
10. end if
11. /* backtrack */
12. logicSimulate_and_imply (pi, ū);
13. result = PODEM-Recursion(C);
14. if result == success then
15.     return(success);
16. end if
17. /* bad decision made at an earlier step, reset pi */
18. logicSimulate_and_imply (pi, X);
19. return(failure);
```

According to the algorithm in PODEM, the search starts by picking an objective, and it backtraces from the objective to a primary input by means of the best path. Controllability measures can be used here to determine which path is regarded as the best. Gradually more primary inputs will be assigned logic values. At any time the target fault becomes unexcited or the *D*-frontier becomes empty, a bad decision must have been made, and reversal of some previous decisions is needed. The backtracking mechanism proceeds by reversing the most recent decision. If reversing the most recent decision also causes a conflict, the recursive algorithm will continue to backtrack to earlier decisions, until no more reversals are possible, at which time the fault is determined to be undetectable.

Three important functions in PODEM-Recursion() are getObjective(), backtrace(), and logicSimulate_and_imply(). The getObjective() function returns the next objective the ATPG should try to justify. Before the target fault has been excited, the objective is simply to set the line on which the target fault resides to the value opposite to the stuck value. Once the fault is excited, the getObjective() function selects the best fault-effect from the *D*-frontier to propagate. The pseudo-code for getObjective() is shown in Algorithm 14.11.

Algorithm 14.11 getObjective(*C*)

```
1. if fault is not excited then
2.     return (g, v̄);
3. end if
4. d = a gate in D-frontier;
5. g = an input of d whose value is X;
6. v = noncontrolling value of d;
7. return (g, v);
```

The backtrace() function returns a primary input assignment from which there is a path of unjustified gates to the current objective. Thus, backtrace() will never traverse through a path consisting of one or more justified gates. From the objective's point of view, the getObjective() function returns an objective, say $g = v$, which means the current value of g is unspecified and should be set to value v. If g was already specified to v, $g = v$ would have never been selected as an objective, because it is already justified. Now, if $g = x$ currently, and the objective is to set $g = v$, there must exist a path of unjustified gates from at least one primary input to g. This backtrace() function can simply be implemented as a loop from the objective to some primary inputs through a path of unspecified values. Algorithm 14.12 shows the pseudo-code for the backtrace() routine.

Finally, the logicSimulate_and_imply() function can simply be a regular logic simulation routine. The added imply is used to derive additional implications, if any, that can enhance the getObjective() routine later on.

Consider the multiplexer circuit shown in Figure 14.28 again. Consider the target fault f stuck-at-0. First, PODEM initializes all gate values to X. Then, the first objective would be to set $f = 1$. The backtrace routine selects $c = 0$ as the decision. After logic simulation, the fault is excited, together with $e = h = 0$. The *D*-frontier at this time is g. The next objective is to advance the *D*-frontier, thus getObjective() returns $a = 1$. Because a is already a primary input, backtrace() will simply return $a = 1$. After simulating $a = 1$, the fault-effect is successfully propagated to the primary output z, and PODEM is finished with this target fault with the computed vector $abc = 1X0$. Table 14.9 shows the series of objectives and backtraces for this example.

Table 14.9 PODEM Objectives and Decisions for *f* Stuck-At-0

getObjective()	backtrace()	logicSim()	D-frontier
$f = 1$	$c = 0$	$d = 0, f = \mathrm{D}, e = 0, h = 0$	g
$a = 1$	$a = 1$	$g = D, z = D$	*f*/0 detected

Algorithm 14.12 backtrace(*C*)

```
1. i = g;
2. num_inversion = 0;
3. while i ≠ primary input do
4.       i = an input of i whose value is X;
5.       if i is an inverted gate type then
6.          num_inversion ++;
7.       end if
8. end while
9. if num_inversion == odd then
10.      v = v̄;
11. end if
12. return (i, v);
```

Consider the circuit shown in Figure 14.29. Suppose the target fault is b stuck-at-0. After circuit initialization, the first objective is $b = 1$ to excite the fault. The backtrace() returns $a = 0$. After logic simulation, although the target fault is excited, there is no *D*-frontier, because $c = d = 0$. At this time, PODEM reverses its last decision $a = 0$ to $a = 1$. After logic simulating $a = 1$, the target fault is not excited and the *D*-frontier is still empty. PODEM backtracks but there is no prior decision point. Thus, it concludes that fault $b/0$ is undetectable. Table 14.10 shows the steps made for this example, and Figure 14.36 shows the corresponding decision tree.

Consider again the circuit shown in Figure 14.34 with the target fault $g/1$. After circuit initialization, the first objective is to excite the fault; in other words, the objective is $g = 0$. The backtrace() function backtraces from the objective backward to a primary input via a path of "don't cares." Suppose the

Table 14.10 PODEM Objectives and Decisions for *b* Stuck-At-0

getObjective()	backtrace()	logicSim()	D-frontier
b = 1	a = 0	b = 1, c = 0, d = 0	{}
a = 1 (reversal)	–	b = 0, c = 1, d = 0	{}

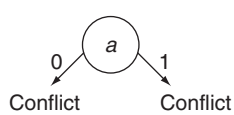


FIGURE 14.36

Decision tree for fault $b/0$.

backtrace reaches $a = 0$. After logic simulation, $g = 0$, $c = d = 0$, and $i = 0$. The *D*-frontier is h. However, note that there is no path of "don't cares" from any fault-effect in the *D*-frontier to a primary output! If the PODEM algorithm is modified to check that any objective has at least a path of "don't cares" to one or more primary outputs, some needless searches can be avoided. For instance, in this example, if the next objective was $f = 1$, even after the decision of $b = 1$ is made, the target fault still would not have been detected, because there was no path to propagate the fault-effect to a primary output even before the decision $b = 1$ was made. In other words, the search could immediately backtrack on the first decision $a = 0$. In this case, $a = 1$, and the objective is still $g = 0$. Backtrace() will now return $b = 0$. After logic simulation, $g = 0$, $c = 1, f = 0, h = 0, i = 0$. Again, there is no propagation path possible. As there is no earlier decision to backtrack to, the ATPG concludes that fault $g/1$ is untestable. Table 14.11 shows the steps for this example.

14.4.5 FAN

Although PODEM reduces the number of decision points from the number of gates in the circuit to the number of primary inputs, it still can make an excessive number of decisions. Furthermore, because PODEM targets one objective at a time, the decision process may sometimes be too localized and miss the global picture. The ***fanout-oriented test generation*** (FAN) algorithm [Fujiwara 1983] extends the PODEM-based algorithm to remedy these shortcomings.

To reduce the number of decision points, FAN first identifies the **headlines** in the circuit, which are the output signals of fanout-free regions. Because of the fanout-free nature of each cone, all signals outside the cone that do not conflict with the headline assignment would never require a conflicting value assignment on the primary inputs of the corresponding fanin cone. In other words, any value assignment on the headline can always be justified by its fanin cone. This allows the backtrace() function to backtrace to either headlines or primary inputs. Because each headline has a corresponding fanin cone with several primary inputs, this allows the number of decision points to be reduced.

Consider the circuit shown in Figure 14.37. If the current objective is to set $z = 1$, the corresponding decision tree based on the PODEM algorithm will involve many decisions at the primary inputs, such as $a = 1$, $c = 1$, $d = 1$, $e = 1, f = 1$. On the other hand, the decision based on the FAN algorithm is

Table 14.11 PODEM Objectives and Decisions for *g* Stuck-At-1

getObjective()	backtrace()	logicSim()	D-frontier
$g = 1$	$a = 0$	$g = D$, $c = 0$, d = 0, $i = 0$	{h} (but no X-path to PO)
$a = 1$ (reversal)	–	$c = 1$, $d = 1$	{}

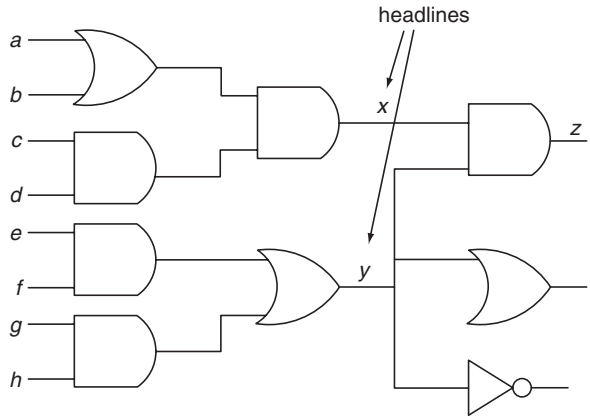


FIGURE 14.37

Circuit with identified headlines.

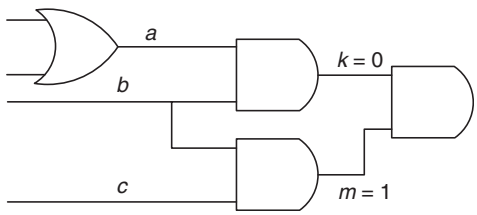


FIGURE 14.38

Multiple backtrace to avoid potential conflicts.

significantly smaller, involving only two decisions: $x = 1$ and $y = 1$. If $z = 1$ was not the first objective, there would have been other decisions made earlier. In other words, if there were a poor decision made in an earlier step, PODEM would need to reverse and backtrack many more decisions compared with FAN.

The next improvement that FAN makes over PODEM is the simultaneous satisfaction of multiple objectives, as opposed to only one target objective at each step. Consider the circuit fragment shown in Figure 14.38. Without taking into account multiple objectives, the backtrace() routine may choose the easier path in trying to justify $k = 0$. The easier path may be through the fanout stem b. However, this would cause a conflict later on with the other objective $m = 1$. In FAN, multiple objectives are taken into account, and the backtrace routine scores the nodes visited from each objective in the current set of objectives. The nodes along the path with the best scores are chosen. In this example, $a = 0$ will be chosen rather than $b = 0$, even if $a = 0$ is less controllable.

A powerful implication engine can have a significant impact on the performance of ATPG algorithms. Thus, much effort has been invested over the years in the efficient computation of implications. The quality of implications was

improved with the computation of indirect implications in **SOCRATES** [Schulz 1988]. **Static learning** was extended to **dynamic learning** in [Schulz 1989 and Kunz 1993], where some nodes in the circuit already had value assignments during the learning process. A 16-valued logic was introduced in [Rajski 1990 and Cox 1994]. Reduction lists were used to dynamically determine the gate values. In [Chakradhar 1993], the authors proposed a transitive closure procedure based on the implication graph. **Recursive learning** was later proposed in [Kunz 1994] in which a complete set of pairwise implications could be computed. To keep the computational costs low, a small recursion depth can be enforced in the recursive learning procedure. Finally, implications to capture time frame information in sequential circuits in a graphical representation were proposed in [Zhao 2001] to compactly store the implications in sequential circuits.

The implications can be used to quickly identify untestable faults [Iyer 1996a,b; Zhao 2001; Hsiao, 2002; Syal 2003]. This will allow the ATPG not to specifically target these faults that can often consume much of the ATPG computational resources. For more information on implication and untestable fault identification, refer to [Bushnell 2000, Jha 2003, and Wang 2006].

14.5 ADVANCED TEST GENERATION

Thus far, the discussions have focused primarily on the basic ATPG algorithms. As circuits have become increasingly larger and more complex, more powerful ATPG algorithms are needed. In particular, the handling of sequential circuits is a must, because not all circuits may have the luxury of having a full-scan inserted. Next, deterministic ATPGs may face tremendous hurdles when dealing with the need to generate a sequence of many vectors. In this regard, simulation-based ATPGs may be better suited. Finally, the stuck-at fault model may be insufficient in capturing defects that occur at the deep-submicron or nanoscale designs. Such defects include delay faults and bridging faults. This section addresses how the basic ATPG can be extended to deal with these issues.

14.5.1 Sequential ATPG: Time frame expansion

Test generation for sequential circuits bears much similarity with that for combinational circuits. However, one vector may be insufficient to detect the target fault, because the excitation and propagation conditions may necessitate some of the flip-flop values to be specified at certain values. The general model for a sequential circuit is shown in Figure 14.39, where flip-flops constitute the memory/state elements of the design. All the flip-flops receive the same clock signal, so no multiple clocks are assumed in the circuit model.

Figure 14.40 illustrates an example of a sequential circuit that is unrolled into several time frames, also called an ***iterative logic array*** (ILA) of the

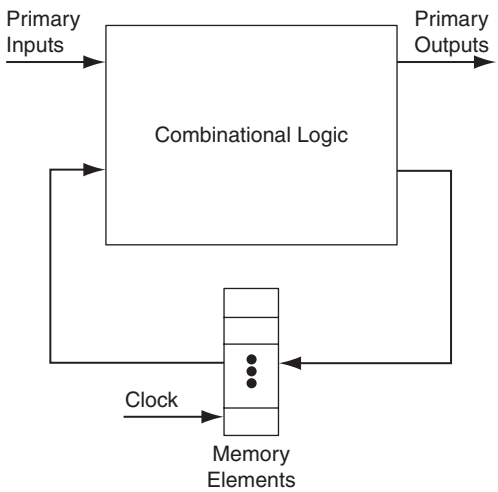


FIGURE 14.39

Model of a sequential circuit.

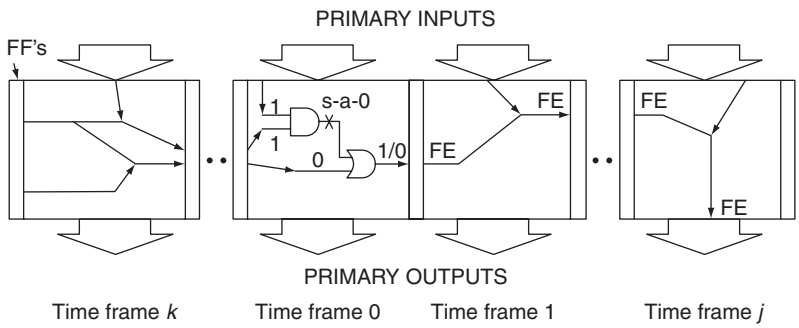


FIGURE 14.40

An iterative logic array (ILA) model.

circuit. For each time frame, the flip-flop inputs from the previous time frame are often referred to as **pseudo primary inputs** with respect to that time frame, and the output signals to feed the flip-flops to the next time frame are referred to as **pseudo primary outputs**. Note that in any unrolled circuit, a target fault is present in every time frame.

When the test generation begins, the first time frame is referred to as time frame 0. An ATPG search similar to a combinational circuit is carried out. At the end of the search, a combinational vector is derived, where the input vector consists of primary inputs and pseudo primary inputs. The fault-effect for the target fault may be sensitized to either a primary output of the time frame or a pseudo primary output. If at least one pseudo primary input has been

specified, then the search must attempt to justify the needed flip-flop values in time frame −1. Similarly, if fault-effects only propagate to pseudo primary outputs, the ATPG must try to propagate the fault-effects across time frame + 1. Note that this results in a **test sequence** of vectors. As opposed to combinational circuits, in which a single vector is sufficient to detect a detectable fault, in sequential circuits a test sequence is often needed.

One question naturally arises: Should the ATPG first attempt the fault excitation via several time frames −1, −2, etc., or should the ATPG attempt to propagate the fault-effect through time frames 1, 2, etc.? It can be observed that in propagating the fault-effect in time frame 1, the search may place additional values on the flip-flops between the boundary of time frames 0 and 1. These added constraints propagate backward and may add additional values needed at the pseudo primary inputs at time frame 0. In other words, if the ATPG first justifies the pseudo primary inputs at time frame 0, it would have missed the additional constraints placed by the propagation. Therefore, the ATPG first tries to propagate the fault-effect to a primary output via several time frames, with all the intermediate flip-flop values propagated back to time frame 0. Then, the ATPG proceeds to justify all the pseudo primary input values at time frame 0.

Although easy to understand, the process can be very complex, for example, if the fault-effect has propagated forward for three time frames: time frames 1, 2, and 3. Now in time frame 4, suppose the ATPG successfully propagates the fault-effect to a primary output (*i.e.*, it has derived a vector at time frame 4), it must go back to time frame 3 to make sure the values assigned to the flip-flops at the boundary between time frames 3 and 4 are, indeed, possible. It must perform this check for time frames 2, 1, and 0. If at any time frame a conflict occurs, the vector derived at time frame 4 is actually invalid, because it is not justifiable from the previous vectors. At this time, a backtrack occurs in time frame 4, and the ATPG must try to find a different solution vector #4. This process is repeated.

One way to reduce the complexity discussed is to try to propagate the fault-effect in an unrolled circuit instead of propagating the fault-effect time frame by time frame. In doing so, a k-frame combinational circuit is obtained, say $k = 256$, and the ATPG views the entire 256-frame circuit as one large combinational circuit. However, the ATPG must keep in mind that the target fault is present in all 256 time frames. This eliminates the need to check for state boundary justifiability and allows the ATPG to propagate the fault-effect across multiple time frames at a time.

When the fault-effect has been propagated to at least one primary output, the pseudo primary inputs at time frame 0 must be justified. Again, the justification can be performed in a similar process of viewing an unrolled 256-frame circuit. As before, the ATPG must ensure that the fault is present in every time frame of the unrolled circuit.

HITEC [Niermann 1991] is a popular sequential test generator that performs the search similar to the discussed methods with a 9-valued algebra. In addition, it uses the concept of **dominators** to help reduce the search complexity. A dominator for a target fault is a gate in the circuit through which

the fault-effect must traverse [Kirkland 1987]. Therefore, for a given target fault, all inputs of any dominator gate that are not in the fanout cone of the fault must be assigned to noncontrolling values to detect the fault.

The concept of controllability and observability metrics can be extended to sequential circuits such that the backtrace routine would prefer to backtrace toward primary inputs and those easy-to-justify flip-flops. The use of sequential testability metrics allows the ATPG to narrow the search space by favoring the easy-to-reach states and avoiding getting into difficult-to-justify states.

The computational complexity of a sequential ATPG is intuitively higher than that of the combinational ATPG. Therefore, aggressive learning can help to reduce the computational cost. For instance, if a known subset of unreachable states is available, this information can be used to allow the ATPG to backtrack much sooner when an intermediate state is unreachable. This can avoid successive justification of an unreachable state. Likewise, if a justification sequence has been successfully computed for state S before, and a different target fault requires the same state *S*, the previous justification sequence can be used to guide the search. Note that, because the target faults are different, the justification sequence may not simply be copied from the solution for one fault to another.

For large circuits, deterministic ATPGs may suffer from a potentially large number of backtracks. Thus, in the past two decades, effort on **simulation-based ATPGS** has yielded much success, presenting themselves as a viable alternative to deterministic ATPGs. One class of nondeterministic ATPGs is the **genetic algorithm–based** (GA-based) ATPG. There have been numerous GA-based ATPGs proposed over the years. For example, **CONTEST** [Agrawal 1989] targets test generation in three phases, each having its own distinct fitness measure. **GATEST** [Rudnick 1994] distinguishes fault detection from those that only propagate to flip-flop boundaries. **DIGATE** [Hsiao 1996] targets individual faults and uses distinguishing sequences to help propagate the faults from flip-flops to a primary output.

STRATEGATE [Hsiao 1997; 2000] addresses fault excitation by justifying the needed state as well. Although GA-based ATPGs have achieved success, the underlying fault simulation engine may incur excessive computational cost. In recent years, approaches that use logic simulation rather than fault simulation have been proposed [Pomeranz 1995; Guo 1999; Giani 2001; Sheng 2002; Wu 2004]. Logic-simulation–based test generators usually target some inherent "property" in the fault-free circuit and try to derive test vectors that exercise these properties. In general, the property used often relates to the states reached by the test sequence.

14.5.2 **Delay fault ATPG**

Today's integrated circuits are seeing an escalating clock rate, shrinking dimensions, increasing chip density, etc. Consequently, there arises a class of defects that would affect the functionality of the design if the chip were run at a high speed.

In other words, the design is functionally correct when it is operated at a slow clock. This type of defect is referred to as a **delay defect**. Although the conventional stuck-at testing can catch some delay defects, the stuck-at fault model is insufficient to model delay defects satisfactorily. This has prompted engineers and researchers to propose a variety of methods and fault models for detecting speed failures. Among the fault models are the *transition fault* [Levendel 1986; Waicukauski 1987; Cheng 1993], the *path-delay fault* [Smith 1985], and the *segment delay fault* [Heragu 1996]. The path-delay fault model considers the cumulative effect of the delays along a specific combinational path in the circuit. If the cumulative delay in a faulty circuit exceeds the clock period for the path, then the test pattern that can exercise this path will fail the chip. The segment delay fault model targets path segments instead of complete paths.

Because a transition has to be launched to propagate across a given path, two vectors are needed. The first vector initializes the circuit nodes, and the second vector launches a transition at the start of a path and ensures that the transition is propagated along the given path. Given a path P, a signal is an **on-input** of *P* if it is on *P*. Conversely, a signal is an **off-input** of *P* if it is an input to a gate in *P* but is not an on-input of *P*. A path-delay fault can be a rising fault, where a rising transition is at the start of the path, or a falling fault, where a falling transition is at the start of the path. The rising and falling path-delay faults are denoted with the up-arrow $\uparrow$ and the down-arrow $\downarrow$ before path P, respectively. For example, $\uparrow g_1 g_4 g_7$ is a rising path that traverses through gates g_1, g_4, and g_7.

Delay tests can be applied three different ways: **launch-on-capture** (also called broad-side [Savir 1994] or double-capture [Wang 2006]), **launch-on-shift** (also called skewed-load [Savir 1993]), and **enhanced-scan** [Dervisoglu 1991]. In launch-on–capture-based testing, the first n-bit vector is scanned into the circuit with n scan flip-flops at a slow speed, followed by another clock that creates the transition. Finally, an at-speed functional clock is applied that captures the response. Thus, only one vector has to be stored per test, and the second vector is directly derived from the initial vector by pulsing the clock. In launch-on–shift-based testing, the first $n - 1$ bits of an n-bit vector are shifted in at a slow speed. The final nth shift is performed, and it is also used to launch the transition. This is followed by an at-speed quick capture. Similar to launch-on-capture, only one vector has to be stored per test, because the second vector is simply the shifted version of the first vector. Finally, in enhanced-scan testing, both vectors in the vector pair (V_1, V_2) have to be stored in the tester memory. The first vector V_1 is loaded into the scan chain, followed by its immediate application to initialize the circuit under test. Next, the second vector is scanned in, followed by an immediate application and capture of the response. Note that the node values in the circuit are preserved during the shifting-in of the second vector V_2. To achieve this, a **hold-scan design** [Dervisoglu 1991] is required.

Because both launch-on-capture and launch-on-shift place constraints on what the second vector can be, they will achieve lower fault coverage compared with enhanced-scan. However, enhanced-scan comes at a price of

hold-scan cells (enhanced-scan cells [Wang 2006]), which consume more chip area. This may not be viewed as a huge negative in microprocessors and some custom-designed circuits, because hold-scan cells are used to prevent the combinational logic from seeing the values being shifted. This is done because the intermediate state of the scan cells may cause contention in some of the signals in the logic, as well as reducing the power consumption in the combinational logic during the shifting of the data in scan cells. In addition, hold-scan cells also help increase the diagnostic capability on failing chips in which the data captured in the scan chain can be retrieved.

In terms of test data volume, enhanced-scan tests may actually require less storage to achieve the same delay fault coverage. In other words, for launch-on-capture or launch-on-shift to achieve the same level of fault coverage, many more patterns may have to be applied.

Unlike stuck-at faults, where a fault is either detected or not detected by a given test vector, a path-delay fault may be detected by different test patterns (consisting of two vectors) with differing levels of quality. In other words, some test patterns can detect a path-delay fault only with certain restrictions in place. Higher quality test patterns place more restrictions on sensitization of the path. On the other hand, similar to stuck-at faults, some paths may be untestable if the sensitization requirement for a given path is not satisfiable.

For designs with two interactive clock domains, modifications can be made to allow for tests. For example, the following at-speed delay test approaches can be used for both launch-on-capture and launch-on-shift architectures: **one-hot double-capture, aligned double-capture**, and **staggered double-capture** [Bhawmik 1997; Wang 2006, 2007b].

If tests were possible for all the paths in a circuit, we would not need any additional test vectors for capturing the delay defects. However, because very few paths are robustly testable, and the number of path-delay faults is exponential to the number of circuit lines, other delay fault models have been proposed. For example, transition tests have been generated to improve the detection of speed failures in microprocessors [Tendulkar 2002], as well as ***application-specific integrated circuits*** (ASICs) [Hsu 2001]. These reasons make transition faults popular in industry.

Similar to the stuck-at fault model, two transition faults are possible at each node of the circuit: *slow-to-rise* and *slow-to-fall*. A test pattern for a transition fault consists of a pair of vectors (V_1, V_2), where V_1 (called the *initial vector*) is required to set the target node to an initial value and V_2 (called the *test vector*) is required to launch the corresponding transition at the target node and also propagate the fault effect to a primary output [Waicukauski 1987; Savir 1993].

Transition tests can also be applied in three different ways as for the other delay fault models discussed earlier: launch-on-capture, launch-on-shift, and enhanced scan. As with path-delay tests, because both launch-on-capture and launch-on-shift place constraints on what the second vector can be, they will achieve lower transition fault coverage compared with enhanced-scan.

14.5.3 Bridging fault ATPG

Recall that bridging faults are those faults that involve a short between two signals in the circuit. Given a circuit with n signals, there are potentially $n \times (n - 1)$ possible bridging faults. However, practically, only those signals that are locally close on the die are more likely to be bridged. Therefore, the total number of bridging faults can be reduced to be linear in the number of signals in the circuit.

Consider two signals x and y in the circuit that are bridged. This bridging fault will not be excited unless different values are placed on x and y. Note that the actual voltage at x and y may be different because of the resistance value of the bridge. Subsequently, the logic that takes x as its input may interpret the logic value differently from the logic that takes y as its input. To reduce the complexity, five common bridging fault models are often used:

1. AND bridge—The faulty value of the bridge for x' and y' is taken to be the logical AND of x and y in the original fault-free circuit.
2. OR bridge—The faulty value of the bridge for x' and y' is taken to be the logical OR of x and y in the original fault-free circuit.
3. x DOM y bridge—x dominates y; in other words, the faulty value of the bridge for both x' and y' is taken to be the logic value of x in the fault-free circuit.
4. x DOM1 y bridge—x dominates y if $x = 1$; in other words, the faulty value of x' is unaffected, but the faulty value for y' is taken to be the logical OR of x and y in the fault-free circuit.
5. x DOM0 y bridge—x dominates y if $x = 0$; in other words, the faulty value of x' is unaffected, but the faulty value for y' is taken to be the logical AND of x and y in the fault-free circuit.

Figure 14.41 illustrates the faulty circuit models corresponding to each of these five bridge types. If a path exists between x and y, then the bridging fault is said to be a **feedback-bridging fault**. Otherwise, it is a **non-feedback-bridging fault**. Figure 14.42 illustrates a feedback-bridging fault. In this figure, if $abc = 110$, then in the fault-free circuit $z = 0$. If the bridge is an AND-bridge, then a cycle would result. In other words, a becomes 0 and in turn makes z = 1. Because $a = 1$ initially, it will again try to drive $z = 0$, resulting in an infinite loop around the bridge. For the following discussion, only non-feedback bridging faults will be considered.

Testing for bridging faults is similar to a constrained stuck-at ATPG. In other words, when testing for the AND-bridge(x, y), either (1) x/0 has to be detected with $y = 0$ or (2) y/0 has to be detected with $x = 0$ [Williams 1973]. A conventional stuck-at ATPG can be modified to handle the added constraint. Likewise, the ATPG can be modified for other bridging fault types.

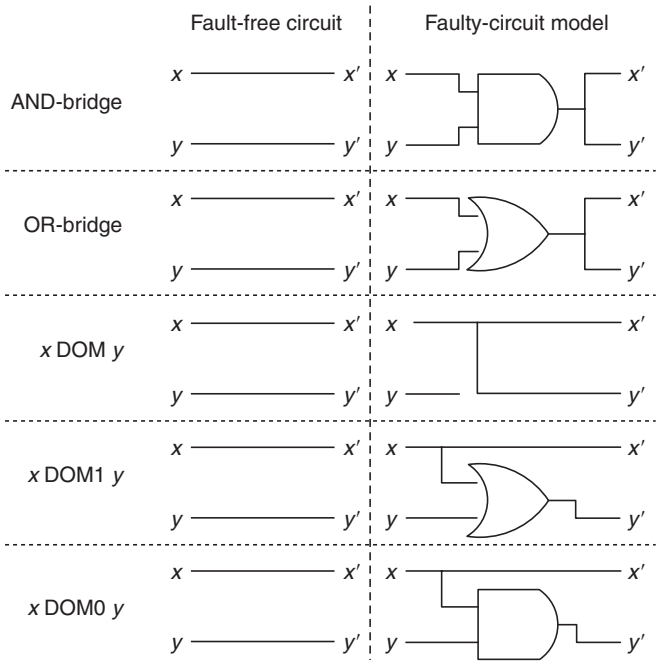


FIGURE 14.41

Bridging fault models.

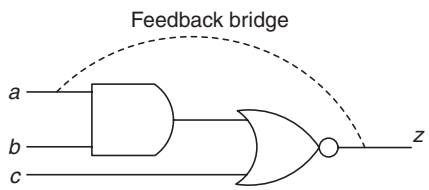


FIGURE 14.42

A feedback bridging fault.

14.6 CONCLUDING REMARKS

For fault simulation, both event-driven simulation and compiled-code simulation techniques can be found in commercially available ***electronic design automation*** (EDA) applications. The fault simulators can be stand-alone tools or used as an integrated feature in the ATPG programs. As a stand-alone tool, concurrent fault simulation with the event-driven simulation technique is used in Verifault-XL (from Cadence Design Systems [Cadence 2008]) and TurboFault (from SynTest Technologies [SynTest 2008]). As an integrated feature in ATPG, bitwise parallel simulation with the compiled-code simulation technique is widely used

in modern commercial ATPG programs, including Encounter Test (from Cadence Design Systems), FastScan (from Mentor Graphics [Mentor 2008]), TetraMAX (from Synopsys [Synopsys 2008]), and TurboScan (from SynTest Technologies).

As we move to the nanometer age, we have started to see nanometer designs that contain hundreds of millions of transistors. We anticipate the semiconductor industry will completely adopt the scan method for quality considerations. As a result, it is becoming imperative that advanced techniques for both logic simulation and fault simulation be developed to address the high-performance and high-capacity issues, in particular, for addressing new fault models, such as transition faults [Waicukauski 1986], path-delay faults [Schulz 1989], bridging faults [Li 2003], and small delay defects [Sato 2005; Hamada 2006]. At the same time, more innovations are needed in developing advanced concurrent fault simulation techniques, because at present designs based on the scan method are still not 100% scan testable. Fault simulation with functional patterns is important for at-speed test applications to detect small delay faults and achieve the ***parts-per-million*** (PPM) defect level goals.

The theory and implementation of an ATPG engine have also been described in detail in the second half of this chapter. Several algorithms were laid out with specific examples given. Advanced ATPG algorithms were discussed where sequential ATPG and ATPG for non-stuck-at faults were covered. Test generation remains to be an important research area as circuit sizes and complexities continue to increase. New and powerful algorithms are needed to cope with the increased complexity. In addition, with nanoscale feature sizes, new defect types and hence new fault models will be needed in future ATPGs.

Should there be defective chips that were uncovered by the test set, fault diagnosis and failure analysis are often subsequently performed to identify the causes and further reduce the defect level in the future. To ease the burden of fault diagnosis and failure analysis, adding ***design-for-debug-and-diagnosis*** (DFD), ***design-for-reliability*** (DFR), ***design-for-manufacturability*** (DFM), and ***design-for-yield*** (DFY) features can be implemented in the design. These features and techniques are extensively discussed in [Wang 2006, 2007a]. Finally, successful ATPG algorithms not only can help in the area of manufacturing tests, but they also provide much insight to other EDA problems, such as synthesis and verification.

14.7 EXERCISES

14.1. (Equivalence Fault Collapsing) How many uncollapsed single stuck-at faults are there in circuit *M* shown in Figure 14.43 Please perform equivalence fault collapsing with the simple_EFC algorithm. How many equivalence collapsed faults do you have?

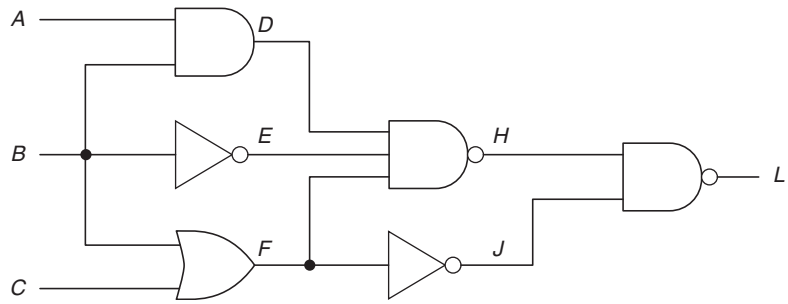


FIGURE 14.43

Circuit *M*.

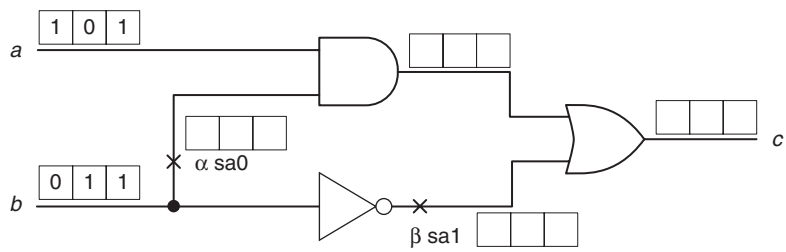


FIGURE 14.44

An example circuit *K*.

14.2. (Dominance Fault Collapsing) Continued from Exercise 14.1. Please perform dominance fault collapsing with the simple_DFC algorithm. How many dominance collapsed faults do you have?

14.3. (Dominance Fault Collapsing) For the circuit in Figure 14.9, please explain why $K/0$ and $K/1$ faults can be removed from the dominance collapsed fault list. Also explain why $F/1$ and $F/0$ can be removed.

14.4. (Parallel-Pattern Single-Fault Propagation) For circuit K shown in Figure 14.44 and two given stuck-at faults shown in Figure 14.44, use the parallel-pattern single-fault propagation fault simulation technique to identify which faults can be detected by the given test patterns.

14.5. (Parallel Fault Simulation) Repeat Exercise 14.4 by use of parallel fault simulation.

14.6. (Concurrent Fault Simulation) Repeat Exercise 14.5 with concurrent fault simulation.

14.7. (Random Test Generation) Given a circuit with three primary outputs, x, y, and z, the fanin cone of x is $\{a, b, c\}$, the fanin cone of y is $\{c, d, e, f\}$, and the fanin cone of z is $\{e, f, g\}$. Devise a pseudo-exhaustive test set for this circuit. Is this test set the minimal pseudo-exhaustive test set?

14.8. (Random Test Generation) With the circuit shown in Figure 14.28, compute the detection probabilities for each of the following faults:
a. $e/0$
b. $e/1$
c. $c/0$

14.9. (Boolean Difference) With the circuit shown in Figure 14.28, compute the set of all vectors that can detect each of the following faults using Boolean difference:
a. $e/0$
b. $e/1$
c. $c/0$

14.10. (Boolean Difference) Assume a single-output combinational circuit, where the output is denoted as f. If two faults, a and b, are indistinguishable, it means that there does not exist a vector that can detect only one and not the other. Show that $f_a \oplus f_b = 0$ if they are indistinguishable.

14.11. (*D* Algorithm) Construct the table for the XNOR operation for the 5-valued logic similar to Tables 14.6, 14.7, and 14.8.

14.12. (*D* Algorithm) Consider a three-input AND gate g. Suppose g is a D-frontier. What are all the possible value combinations the three inputs of g can take such that g is a valid D-frontier?

14.13. (PODEM) With the circuit shown in Figure 14.28, compute a test vector that can detect each of the following faults by use of PODEM:
a. $e/0$
b. $e/1$
c. $c/0$

14.14. (FAN) Consider the circuit shown in Figure 14.37. Suppose the constraint that $y = 1 \rightarrow x = 0$ is given. How could one use this knowledge to reduce the search space when trying to generate vectors in the circuit? For example, suppose the target fault is $y/0$.

14.15. (Sequential ATPG) Consider the circuit shown in Figure 14.45. The target fault is $a/0$.

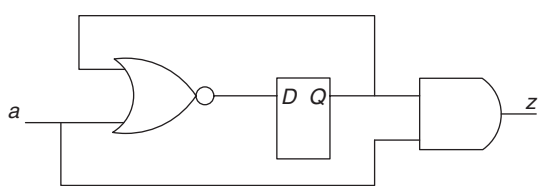


FIGURE 14.45

Example sequential circuit.

a. Generate a test sequence for the target fault by use of only 5-valued logic.
b. Generate a test sequence for the target fault by use of 9-valued logic.

14.16. (Sequential ATPG) Given a sequential circuit, is it possible that two stuck-at faults, $a/0$ and $a/1$, are both detected by the same vector v_i in a test sequence $v_0, v_1, \ldots, v_k$?

14.17. (Sequential ATPG) Consider an **iterative logic array** (ILA) expansion of a sequential circuit, where the initial pseudo primary inputs are fully controllable. Show that the states reachable in successive time frames of the ILA shrink monotonically.

14.18. (Bridging Faults) Consider a bridging fault between the outputs of an AND gate $x = ab$ and an OR gate $y = c + d$. What values to $abcd$ would induce the largest current in the bridge?

ACKNOWLEDGMENTS

We thank Professor Hank Walker of the University of A&M for contributing a portion of the Fault Simulation section; and Professor Xiaoqing Wen of Kyushu Institute of Technology and Professor Charles E. Stroud of Auburn University for reviewing the text and providing helpful comments.

REFERENCES

R14.0 Books

[Abramovici 1994] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design, Revised Printing,* IEEE Press, Piscataway, NJ, 1994.

[Bushnell 2000] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI circuits,* Springer Science, New York, 2000.

[Holland 1975] J. H. Holland, *Adaptation in Natural and Artificial Systems,* University of Michigan Press, Ann Arbor, MI, 1975.

[Jha 2003] N. Jha and S. Gupta, *Testing of Digital Systems,* Cambridge University Press, London, 2003.

[Wang 2006] L.-T. Wang, C.-W. Wu, and X. Wen, editors, *VLSI Test Principles and Architectures: Design for Testability,* Morgan Kaufmann, San Francisco, 2006.

[Wang 2007a] L.-T. Wang, C. E. Stroud, and N. A. Touba, editors, *System-on-Chip Test Architectures: Nanometer Design for Testability,* Morgan Kaufmann, San Francisco, November 2007.

R14.1 Fault Collapsing

[McCluskey 1971] E. J. McCluskey and F. W. Clegg, Fault equivalence in combinational logic networks, *IEEE Trans. on Computers*, C-20(11), pp. 1286–1293, November 1971.

R14.2 Fault Simulation

[Abramovici 1984] M. Abramovici, P. R. Menon, and D. T. Miller, Critical path tracing: An alternative to fault simulation, *IEEE Design & Test of Computers*, 1(1), pp. 83–93, February 1984.

[Butler 1974] T. T. Butler, T. G. Hallin, J. J. Kulzer, and K. W. Johnson, LAMP: Application to switching system development, *Bell System Technical J.*, 53, pp. 1535–1555, October 1974.

[Cheng 1989] W. T. Cheng and M. L. Yu, Differential fault simulation: A fast method using minimal memory, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 424–428, June 1989.

[Goel 1980] P. Goel, Test generation cost analysis and projections, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 77–84, June 1980.

[Jain 1985] S. K. Jain and V. D. Agrawal, Statistical fault analysis, *IEEE Design & Test of Computers*, 2(1), pp. 38–44, February 1985.

[Niermann 1992] T. M. Niermann, W.-T. Cheng, and J. H. Patel, PROOFS: A fast, memory-efficient sequential circuit fault simulator, *IEEE Trans. on Computer-Aided Design*, 11(2), pp. 198–207, February 1992.

[Schulz 1989] M. Schulz, F. Fink, and K. Fuchs, Parallel pattern fault simulation of path delay faults, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 357–363, June 1989.

[Seshu 1965] S. Sesuh and D. N. Freeman, On improved diagnosis program, *IEEE Trans. on Electronic Computers*, Vol. EC-14(1), pp. 76–79, February 1965.

[Ulrich 1974] E. G. Ulrich and T. Baker, Concurrent simulation of nearly identical digital networks, *IEEE Trans. on Computers*, 7(4), pp. 39–44, April 1974.

[Waicukauski 1985] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy, Fault Simulation for Structured VLSI, in *Proc. VLSI System Design*, 6(12), pp. 20–32, December 1985.

[Waicukauski 1986] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, Transition fault simulation by parallel pattern single fault propagation, in *Proc. IEEE Int. Test Conf.*, pp. 542–549, September 1986.

R14.3 Test Generation

[Breuer 1971] M. A. Breuer, A random and an algorithmic technique for fault detection test generation for sequential circuits, *IEEE Trans. on Computers*, 20(11), pp. 1364–1370, November 1971.

[Chakradhar 1993] S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler, A transitive closure algorithm for test generation, *IEEE Trans. on Computer-Aided Design*, 12(7), pp. 1015–1028, July 1993.

[Cox 1994] H. Cox and J. Rajski, On necessary and non-conflicting assignments in algorithmic test pattern generation, *IEEE Trans. on Computer-Aided Design*, 13(4), pp. 515–530, April 1994.

[David 1976] R. David and G. Blanchet, About random fault detection of combinational networks, *IEEE Trans. on Computers*, C-25(6), pp. 659–664, June 1976.

[Fujiwara 1983] H. Fujiwara and T. Shimono, On the acceleration of test generation algorithms, *IEEE Trans. on Computers*, C-32(12), pp. 1137–1144, December 1983.

[Goel 1981] P. Goel, An implicit enumeration algorithm to generate tests for combinational logic circuits, *IEEE Trans. on Computers*, C-30(3), pp. 215–222, March 1981.

[Hsiao 2002] M. S. Hsiao, Maximizing impossibilities for untestable fault identification, in *Proc. Design, Automation, and Test in Europe Conf.*, pp. 949–953, March 2002.

[Iyer 1996a] M. A. Iyer and M. Abramovici, FIRE: A fault independent combinational redundancy algorithm, *IEEE Trans. VLSI Syst*, 4(2), pp. 295–301, June 1996.

[Iyer 1996b] M. A. Iyer, D. E. Long, and M. Abramovici, Identifying sequential redundancies without search, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 457–462, June 1996.

[Kunz 1993] W. Kunz and D. K. Pradhan, Accelerated dynamic learning for test pattern generation in combinational circuits, *IEEE Trans. on Computer-Aided Design*, 12(5), pp. 684–694, May 1993.

[Kunz 1994] W. Kunz and D. K. Pradhan, Recursive learning: A new implication technique for efficient solutions to CAD problems—test, verification, and optimization, *IEEE Trans. on Computer-Aided Design*, 13(9), pp. 1149–1158, September 1994.

[Lisanke 1987] R. Lisanke, F. Brglez, A. J. Degeus, and D. Gregory, Testability-driven random test-pattern generation, *IEEE Trans. on Computer-Aided Design*, 6(6), pp. 1082–1087, November 1987.

[Marques-Silva 1999] J. P. Marques-Silva and K. A. Sakallah, GRASP: A search algorithm for propositional satisfiability, *IEEE Trans. on Computers*, 48(5), pp. 506–521, May 1999.

[Muth 1976] P. Muth, A nine-valued circuit model for test generation, *IEEE Trans. on Computers*, C-25(6), pp. 630–636, June 1976.

[Rajski 1990] J. Rajski and H. Cox, A method to calculate necessary assignments in ATPG, in *Proc. IEEE Int. Test Conf.*, pp. 25–34, October 1990.

[Roth 1966] J. P. Roth, Diagnosis of automata failures: A calculus and a method, in *IBM J. Research and Development*, 10(4), pp. 278–291, July 1966.

[Roth 1967] J. P. Roth, W. G. Bouricius, and P. R. Schneider, Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits, *IEEE Trans. on Electron. Comput.*, EC-16(10), pp. 567–579, October 1967.

[Schnurmann 1975] H. D. Schnurmann, E. Lindbloom, and R. G. Carpenter, The weighted random test-pattern generator, *IEEE Trans. on Computers*, 24(7), pp. 695–700, July 1975.

[Schulz 1988] M. H. Schulz, E. Trischler, and T. M. Sarfert, SOCRATES: A highly efficient automatic test pattern generation system, *IEEE Trans. on Computer-Aided Design*, 7(1), pp. 126–137, January 1988.

[Schulz 1989] M. H. Schulz and E. Auth, Improved deterministic test pattern generation with applications to redundancy identification, *IEEE Trans. on Computer-Aided Design*, 8(7), pp. 811–816, July 1989.

[Seshu 1965] S. Seshu and D. N. Freeman, The diagnosis of synchronous sequential switching systems, *IEEE Trans. on Electron. Comput.*, 11, pp. 459–465, August 1962.

[Shedletsky 1977] J. J. Shedletsky, Random testing: Practicality vs. verified effectiveness, in *Proc. IEEE Int. Symp. on Fault-Tolerant Computing*, pp. 175–179, June 1977.

[Syal 2003] M. Syal and M. S. Hsiao, A novel, low-cost algorithm for sequentially untestable fault identification, in *Proc. ACM/IEEE Design, Automation, and Test in Europe Conf.*, pp. 316–321, March 2003.

[Zhao 2001] J. Zhao, J. A. Newquist, and J. H. Patel, A graph traversal based framework for sequential logic implication with an application to C-cycle redundancy identification, in *Proc. IEEE Int. Conf. on VLSI Design*, pp. 163–169, January 2001.

R14.4 Advanced Test Generation

[Agrawal 1989] V. D. Agrawal, K.-T. Cheng, and P. Agrawal, A directed search method for test generation using a concurrent simulator, *IEEE Trans. on Computer-Aided Design*, 8(2), pp. 131–138, February 1989.

[Bhawmik 1997] S. Bhawmik, Method and Apparatus for Built-In Self-Test with Multiple Clock Circuits, U.S. Patent No. 5,680,543, October 21, 1997.

[Cheng 1993] K.-T. Cheng, S. Devadas, and K. Keutzer, Delay-fault test generation and synthesis for testability under a standard scan design methodology, *IEEE Trans. on Computer-Aided Design*, 12(8), pp. 1217–1231, August 1993.

[Dervisoglu 1991] B. Dervisoglu and G. Stong, Design for testability: Using scanpath techniques for path-delay test and measurement, in *Proc. IEEE Int. Test Conf.*, pp. 365–374, October 1991.

[Giani 2001] A. Giani, S. Sheng, M. S. Hsiao, and V. Agrawal, Efficient spectral techniques for sequential ATPG, in *Proc. IEEE Design, Automation, and Test in Europe Conf.*, pp. 204–208, March 2001.

[Guo 1999] R. Guo, S. M. Reddy, and I. Pomeranz, Proptest: A property based test pattern generator for sequential circuits using test compaction, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 653–659, June 1999.

[Heragu 1996] K. Heragu, J. H. Patel, and V. D. Agrawal, Segment delay faults: A new fault model, in *Proc. IEEE VLSI Test Symp.*, pp. 32–39, April 1996.

[Hsiao 1996] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, Automatic test generation using genetically engineered distinguishing sequences, in *Proc. IEEE VLSI Test Symp.*, pp. 216–223, April 1996.

[Hsiao 1997] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, Sequential circuit test generation using dynamic state traversal, in *Proc. European Design and Test Conf.*, pp. 22–28, February 1997.

[Hsiao 2000] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, Dynamic state traversal for sequential circuit test generation, *ACM Trans. on Design Automation of Electronic Systems*, 5(3), pp. 548–565, July 2000.

[Hsu 2001] F. F. Hsu, K. M. Butler, and J. H. Patel, A case study of the Illinois scan architecture, in *Proc. IEEE Int. Test Conf.*, pp. 538–547, October 2001.

[Kirkland 1987] T. Kirkland and M. R. Mercer, A topological search algorithm for ATPG, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 502–508, June 1987.

[Levendel 1986] Y. Levendel and P. Menon, Transition faults in combinational circuits: Input transition test generation and fault simulation, in *Proc. Fault-Tolerant Computing Symp.*, pp. 278–283, July 1986.

[Niermann 1991] T. M. Niermann and J. H. Patel, HITEC: A test generation package for sequential circuits, in *Proc. European Design Automation Conf.*, pp. 214–218, February 1991.

[Pomeranz 1995] 1Pomeranz and S. M. Reddy, LOCSTEP: A logic simulation based test generation procedure, in *Proc. Fault-Tolerant Computing Symp.*, pp. 110–119, June 1995.

[Rudnick 1994] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann, Sequential circuit test generation in a genetic algorithm framework, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 698–704, June 1994.

[Savir 1993] J. Savir and S. Patil, Scan-based transition test, *IEEE Trans. on Computer-Aided Design*, 12(8), pp. 1232–1241, August 1993.

[Savir 1994] J. Savir and S. Patil, On broad-side delay test, in *Proc. IEEE VLSI Test Symp.*, pp. 284–290, April 1994.

[Sheng 2002] S. Sheng, K. Takayama, and M. S. Hsiao, Effective safety property checking based on simulation-based ATPG, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 813–818, June 2002.

[Smith 1985] G. L. Smith, Model for delay faults based upon paths, in *Proc. IEEE Int. Test Conf.*, pp. 342–349, October 1985.

[Tendulkar 2002] N. Tendulkar, R. Raina, R. Woltenburg, X. Lin, B. Swanson, and G. Aldrich, Novel techniques for achieving high at-speed transition fault coverage for Motorola's microprocessors based on PowerPC instruction set architecture, in *Proc. IEEE VLSI Test Symp.*, pp. 3–8, April 2002.

[Waicukauski 1987] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, Transition fault simulation, *IEEE Design & Test of Computers*, 4(2), pp. 32–38, April 1987.

[Wang 2007b] L.-T. Wang, P.-C. Hsu, and X. Wen, Multiple-Capture DFT System for Detecting or Locating Crossing Clock-Domain Faults During Scan-Test, U.S. Patent No. 7,260,756, August 21, 2007.

[Williams 1973] M. J. Y. Williams and J. B. Angel, Enhancing testability of large-scale integrated circuits via test points and additional logic, *IEEE Trans. on Computers*, C-22(1), pp. 46–60, January 1973.

[Wu 2004] Q. Wu and M. S. Hsiao, Efficient ATPG for design validation based on partitioned state exploration histories, in *Proc. IEEE VLSI Test Symp.*, pp. 389–394, April 2004.

R14.5 Concluding Remarks

[Cadence 2008] Cadence Design Systems, http://www.cadence.com, April 2004.

[Hamada 2006] S. Hamada, T. Maeda, A. Takatori, Y. Noduyama, and Y. Sato, Recognition of sensitized longest paths in transition-delay test, in *Proc. IEEE Int. Test Conf.*, Paper 11.1, October 2006.

[Li 2003] Z. Li, X. Lu, W. Qiu, W. Shi, and D. M. H. Walker, A circuit level fault model for resistive bridges, *ACM Trans. on Design Automation of Electronic Systems*, 8(4), pp. 546–559, October 2003.

[Mentor 2008] Mentor Graphics, http://www.mentor.com, 2008.

[Sato 2005] Y. Sato, S. Hamada, T. Maeda, A. Takatori, Y. Nozuyama, and S. Kajihara, Invisible delay quality-SDQM model lights up what could not be seen, in *Proc. IEEE Int. Test Conf.* Paper 47.1, November 2005.

[Synopsys 2008] Synopsys, http://www.synopsys.com, October 2003.

[SynTest 2008] SynTest Technologies, http://www.syntest.com, 2008.

[Waicukauski 1986] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, Transition fault simulation by parallel pattern single fault propagation, in *Proc. IEEE Int. Test Conf.*, pp. 542–549, September 1986.